Desenvolvimento web com

ASP.NET MVC



Casa do Código

FABRÍCIO SANCHEZ
MÁRCIO FÁBIO ALTHMANN

Casa do Código
Livros para o programador
Rua Vergueiro, 3185 - 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil

Agradecimentos

Uma obra como esta é sempre fruto do esforço direto ou indireto de muitas pessoas. Assim, nada mais justo que dedicar algumas poucas, mas expressivas, palavras a elas.

Fabrício Lopes Sanchez

A Deus, pela capacidade intelectual mínima para realizar um projeto como este.

À minha família, por atuarem como luzes quando tudo em volta se apaga.

Ao Márcio Fabio Althmann (muito mais do que um profissional de renome, um amigo), pela parceria neste projeto.

À editora Casa do Código, por acreditar no projeto e fornecer todo suporte de que precisamos durante todo o processo de composição desta obra.

Márcio Fabio Althmann

A toda minha família e amigos.

Em especial à minha esposa Juliane Mateus Costa. Sempre me apoiando e dando forças para encarar e vencer os obstáculos.

Ao Fabrício Sanchez que felizmente aceitou construir comigo esse projeto.

Sobre os autores

Este livro é o resultado final de anos de estudo e acúmulo de experiência de dois profissionais do mercado de tecnologia, mais especificamente, do volátil e evolutivo universo de desenvolvimento de software. Esta seção é dedicada a você que deseja saber mais sobre os autores desta obra.

Fabrício Lopes Sanchez

Fabrício Lopes Sanchez é graduado em Ciência da Computação e mestre em Processamento Digital de Sinais. Com mais de 15 anos de experiência no mercado de desenvolvimento de softwares, Fabrício acumulou conhecimentos sólidos em diferentes plataformas tecnológicas (C/C++, Java, PHP), embora tenha se especializado em desenvolvimento para web na plataforma Microsoft. Desenvolvedor, arquiteto e diretor de tecnologia de uma startup (cargo que exerce nos dias atuais) são algumas das funções exercidas por Fabrício nos últimos anos. Atuou também como especialista técnico de projetos de computação em nuvem na Microsoft Brasil, onde pôde participar de grandes projetos utilizando a plataforma da empresa para este fim (a saber, Windows Azure). Em 2011, Fabrício foi nomeado pela Microsoft como :Most Valuable Professional: (MVP) na categoria ASP.NET/IIS, prêmio que a empresa concede a profissionais aos quais ela julga destaques em suas tecnologias.

Fabrício escreve de forma recorrente em seu blog: http://fabriciosanchez.com.br. Além disso, ele pode ser encontrado também no Twitter através do usuário @SanchezFabricio.

Márcio Fabio Althmann

Márcio Fábio Althmann possui mais de 10 anos de experiência no mercado de desenvolvimento de software e é especialista na plataforma .NET.

Coautor do livro *Desenvolvendo para web usando o Visual Studio 2008* no qual

abordou o tema Explorando o Acesso a Dados Utilizando LINQ To SQL.

Vencedor do concurso WinThe7 realizado pela Microsoft Brasil, na categoria Desenvolvimento. Graças ao concurso teve a oportunidade de conhecer a sede da Microsoft em Redmond nos EUA.

Atualmente trabalha na equipe de tecnologia da Benner Sistemas, equipe que é responsável por desenvolver as ferramentas utilizadas pelos desenvolvedores da empresa.

Blog: http://www.marcioalthmann.net GitHub: http://github.com/marcioalthmann Twitter: @marcialthmann

Prefácio

A plataforma ASP.NET continua a evoluir e expandir, acompanhando a web. Quando o modelo *web forms* de desenvolvimento foi introduzido, ele trouxe à programação web desenvolvedores que já possuíam ampla experiência com desenvolvimento para *desktops*. O ASP.NET *web forms* coloca uma camada de abstração sobre o protocolo HTTP, possibilitando assim um modelo de desenvolvimento baseado em eventos. Algo mais familiar para estes desenvolvedores. Como a web se modificou e introduziu novas técnicas, *tags* e apresentou o amplo poder do JavaScript, a plataforma ASP.NET precisou acompanhar tal evolução.

O *framework* ASP.NET MVC valoriza coisas diferentes do ASP.NET *web forms*. Enquanto aplicações ASP.NET *web forms* possibilitam a criação de interfaces ricas utilizando HTML5 e JavaScript utilizando um modelo baseado em controles de servidor vinculados a trechos de códigos específicos, o ASP.NET MVC disponibiliza o mesmo poder de forma diferente, trazendo o desenvolvedor mais para "perto do metal". Aplicações web baseadas no *framework* MVC nos dá o controle absoluto de todos os elementos HTML e de cada byte a fio. Aplicações MVC exigem que desenvolvedores conheçam mais sobre web, pois ele permite "abraçar" cada detalhe dela.

Todos as aplicações construídas com a tecnologia ASP.NET — Web Forms e MVC — são baseadas no mesmo núcleo. Cada um destes modelos compartilha conceitos comuns, como diretórios padrão, editores modelos e o repositórios padrão. Para entender de forma assertiva o ASP.NET e com todos os seus aspectos, é preciso recorrer a manuais e livros que possam ajudar. É aí que livros como este entram em cena.

Como a ampla adoção do ASP.NET MVC em todo o mundo, os livros, como o que você está segurando, são muito mais importantes. O Fabrício e o Márcio montaram um excelente guia técnico, que lhe dará toda a informação necessária para expandir sua compreensão do ASP.NET MVC bem como conceitos maiores que cercam a plataforma ASP.NET. À medida que avançamos para melhor unificar a plata-

forma ASP.NET, você certamente vai querer ter uma visão clara e profunda do que ASP.NET MVC tem para oferecer. Eu espero que você se divirta com volume de aprendizado sobre ASP.NET que o espera neste livro. Aproveite o livro do Fabricio e do Márcio.

Scott Hanselman
Principal Program Manager – Azure and Web Tools
Microsoft Corp.

Sumário

Capítulo 1

Sua primeira aplicação

Há alguns anos, a web era um ambiente lúdico. Esta afirmação pode ser facilmente comprovada ao identificarmos o principal objetivo dos usuários daquele período: "entreter". Era muito comum ouvir afirmações do tipo: "Internet? Isso é coisa de desocupado que não tem o que fazer!". Evidentemente, o que faz algo acontecer de fato no mercado é a demanda e, para a demanda daquele momento, as tecnologias disponíveis (HTML, JavaScript e uma linguagem de programação do server-side) eram suficientes. Destacavam-se naquele momento como linguagens server-side: PHP, ASP, CGI, Java (Servlets e posteriormente, JSP) e outras.

O tempo passou e a internet deixou de ser um ambiente estritamente voltado para o entretenimento e passou a ser um ambiente também de negócios. Evidentemente que o perfil do usuário também sofreu alterações. O usuário que antes acessava um website apenas para ler suas notícias (por exemplo), agora acessava websites também para consultar preços de produtos, reservar passagens aéreas, solicitar orçamentos para serviços etc. É desnecessário mencionar aqui que uma nova demanda havia sido criada e que, os websites, passaram a ter traços de aplicações (por inércia,

com maior complexidade associada).

Falando especificamente da Microsoft, com esta nova demanda do mercado por "aplicações web", eis que surge em 2002 o ASP.NET, trazendo consigo o modelo WebForms de programar e estruturar as *web apps*. Sim, naquela época os WebForms causaram espanto. Com o desenvolvimento das aplicações totalmente voltado para a manipulação de componentes do lado servidor (TextBox, GridView, DropDownList, Label etc.) e a facilidade de injeção de comportamentos destes através de seus eventos proporcionada pelo Visual Studio (arrasta o componente, duplo clique no mesmo e inserção de código no evento), a Microsoft arrebanhou uma grande fatia de desenvolvedores, principalmente aqueles já acostumados com esse modelo ("Delphistas" e "VBistas"). Assim, as aplicações web tornaram-se corporativistas, fato este que agradou o mercado e resultou em uma grande adoção da plataforma tecnológica (a.k.a, .NET).

Já para os desenvolvedores web tradicionais, acostumados com o a manipulação direta dos elementos HTML, JavaScript e linguagens server side, o ASP.NET WebForms apresentou-se como um ser completamente estranho, principalmente pelo fato de "tirar" do desenvolvedor o controle total dos elementos citados acima. Ganhava-se em produtividade e familiaridade, entretanto, perdia-se em essência. Na verdade, para estes, a impressão que os WebForms causavam era: "isso não é web".

Olhando através deste prisma e também o antigo e funcional modelo de desenvolvimento (proposto para utilização com a linguagem Smalltalk), o modelo MVC (*Model-View-Controller*), a Microsoft apresentou em 2007, à comunidade técnica, a primeira versão pública de seu novíssimo *framework* para desenvolvimento de aplicações web, o ASP.NET MVC. Na ocasião em que este livro foi escrito, a tecnologia encontrava-se na versão 4.

1.1 E então, vamos começar?

Neste ponto, algumas informações básicas (mas fundamentais, é importante observar) necessárias para seguir com os estudos acerca do modelo MVC de desenvolvimento já se encontram reunidas. Estamos aptos, portanto, a subir o primeiro degrau nesta interessante e divertida escada do conhecimento. Criaremos, assim, nosso primeiro projeto ASP.NET MVC.

O que você precisará para executar os exemplos deste livro?

Para seguir os exemplos apresentados neste livro, você deverá possuir alguns ele-

mentos de software devidamente instalados e configurados em seu computador. Para tornar o entendimento mais claro, vamos segmentar a necessidade de recursos conforme a demanda do exemplo, ok? A lista a seguir, apresenta os requisitos necessários para a composição do primeiro exemplo:

- Microsoft Visual Studio Express 2012 para web (http://bit.ly/mvc-vsexpress) ;
- NET framework 4.5 (Visual Studio já incorpora esta instalação);
- Você encontrará um tutorial sobre como realizar a instalação destes recursos no apêndice deste livro.

Sua primeira aplicação

Criaremos uma aplicação ASP.NET MVC simples que, evidentemente, exibirá uma mensagem de texto "Olá mundo!" (provavelmente seriamos hostilizados se fosse diferente). O objetivo aqui é apresentar o Visual Studio e as ferramentas lá disponíveis para se trabalhar com ASP.NET MVC.

> **Já conhece o Visual Studio?**
>
> Visual Studio (VS) é a principal ferramenta da Microsoft para o desenvolvimento de aplicações. Atualmente, a ferramenta encontra-se na versão 2012. Visual Studio é um produto e como tal, é pago. Entretanto, a Microsoft disponibiliza uma versão gratuita (Express Web), que muito embora possua um número mais reduzido de recursos, traz tudo o que precisamos para construir aplicações web consistentes e eficientes.
>
> Você pode encontrar mais informações sobre o Visual Studio seguindo o link: http://bit.ly/mvc-visualstudio.

Com o Visual Studio 2012 em execução, navegue até a opção `File > New > Project`. Nossa intenção é criar uma nova aplicação para web — desta forma, na janela que se apresentará, você deverá selecionar do lado esquerdo a opção "web" e na sequência (lado direito), "ASP.NET MVC Web Application". A figura 1.1 apresenta a janela com a seleção mencionada.

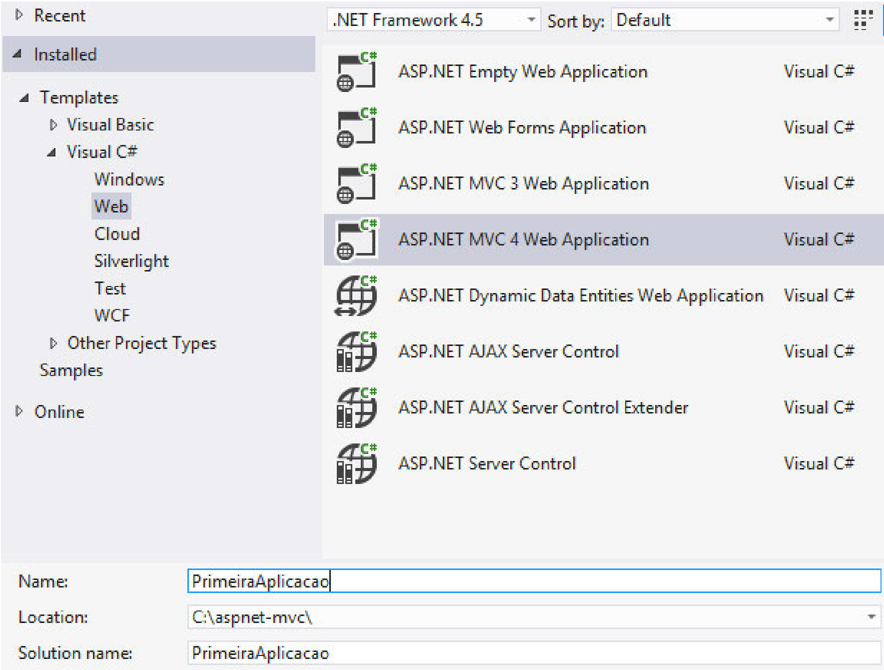


Figura 1.1: Selecionando o tipo de projeto ASP.NET

Tecnologias utilizadas

Apesar de C# não ser a única linguagem disponível em .NET, ela será a linguagem utilizada em todos os exemplos desse livro. Os exemplos também utilizarão a última versão do .NET, que é o 4.5.

Na sequência, você precisará escolher o tipo de projeto (figura 1.2) ASP.NET MVC que deseja criar. Isto porque, acoplado ao Visual Studio, a Microsoft disponibiliza alguns *templates* pré-construídos de projetos para ajudá-lo a endereçar a demanda da aplicação e, claro, ajudá-lo a "poupar trabalho" na estruturação do projeto, trazendo, por exemplo, plugins normalmente utilizados (jQuery, OData, EntityFramework etc.). Para a construção deste primeiro exemplo, utilizaremos o modelo *Internet Application*.

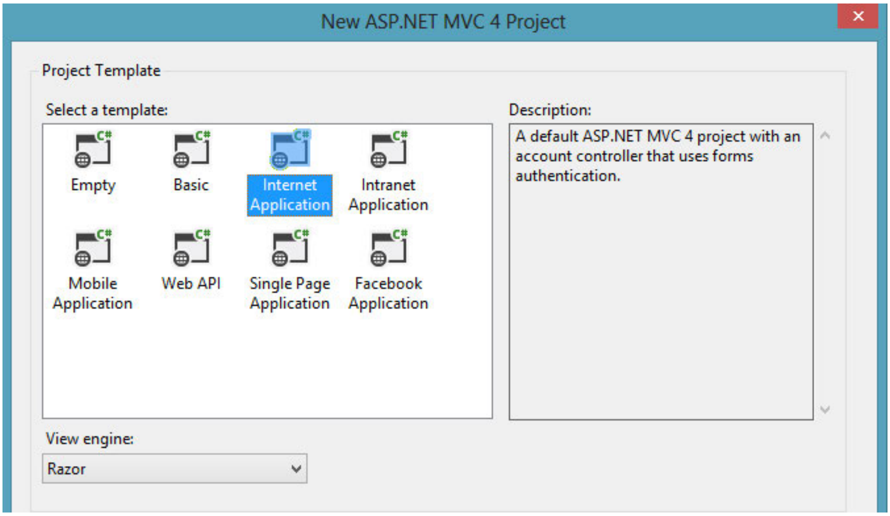


Figura 1.2: Selecionando o tipo de projeto ASP.NET MVC

Após a criação do projeto, você verá no "Solution Explorer" (janela onde os arquivos do projeto são dispostos) do seu Visual Studio, uma estrutura de projeto parecida com aquela apresentada pela figura 1.3. Não se preocupe agora com os detalhes estruturais do projeto, eles serão explicados em detalhes nos próximos capítulos.

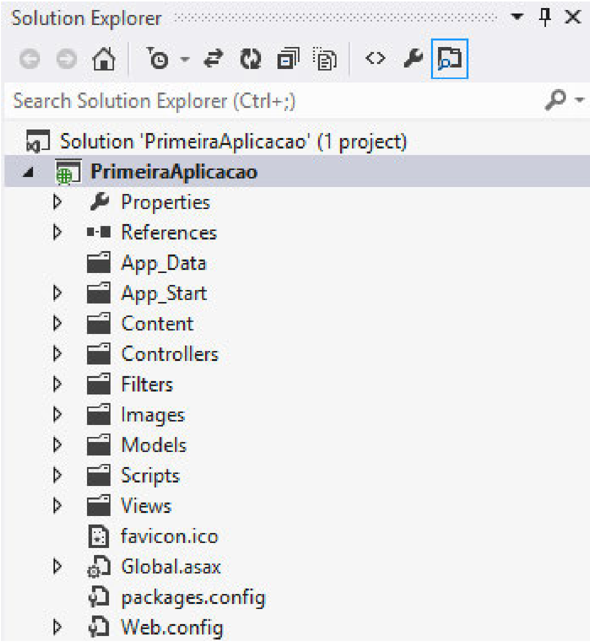


Figura 1.3: Estrutura inicial do projeto

Muito embora o Visual Studio já crie uma estrutura de projeto funcional, antes de executar a aplicação pela primeira vez, faremos uma pequena modificação no corpo da mesma. No Solution Explorer, navegue até o arquivo "HomeController" (`SeuProjeto > Controllers > HomeController`). No método Index, faremos uma alteração simples: modificaremos uma mensagem que é exibida por padrão, na página inicial. Modifique o texto atribuído à propriedade "ViewBag.Message" para "**Minha primeira aplicação com ASP.NET MVC**", conforme apresentado pela listagem 1.

Listagem 1.1 - Alterando o texto de exibição na página inicial:

```
public ActionResult Index()
{
    ViewBag.Message = "Minha primeira aplicação com ASP.NET MVC";

    return View();
}
```

Compile o projeto (você pode utilizar a tecla F5 como atalho para executar esta tarefa) e quando o site for carregado em seu navegador, a mensagem que acabamos de adicionar deverá ser exibida, a exemplo do que apresenta a figura 1.4.

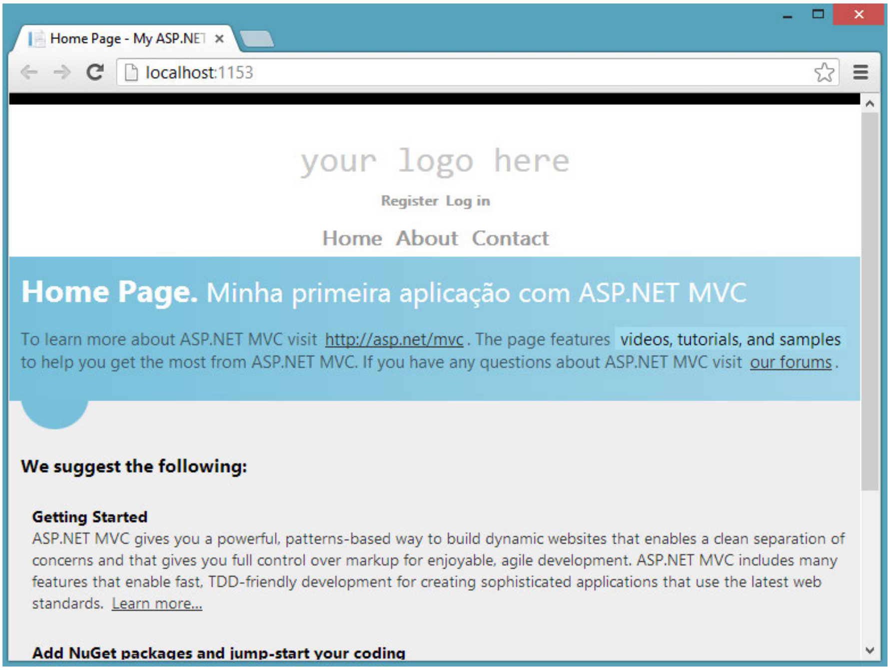


Figura 1.4: Primeira aplicação em funcionamento

Parabéns! Você acabou de criar e "publicar" (localmente, é claro), sua primeira aplicação ASP.NET MVC.

Conforme mencionado anteriormente, você não deve se preocupar neste momento com os aspectos arquiteturais da aplicação, tampouco com nomenclaturas e convenções de código. A ideia aqui é mostrar a você uma aplicação ASP.NET MVC funcionando.

Recomendamos que acompanhe atentamente cada tópico do novo capítulo, uma vez que ele trará detalhes preciosos escondidos por trás de um simples "Minha primeira aplicação com ASP.NET MVC".

Desafio: Agora que você já sabe como é simples criar um projeto ASP.NET MVC utilizando o Visual Studio 2012, gostaria de lançar um desafio antes de partir para a leitura do próximo capítulo. O desafio é: Crie um novo projeto ASP.NET MVC

do tipo "Internet Application" e, no lugar da mensagem "Minha primeira aplicação ASP.NET MVC", adicione a mensagem "**Hoje é: {Data atual}**" e, em seguida, compile e execute o projeto.

1.2 Código fonte e lista de discussão

O código fote da aplicação que estamos construindo no livro pode ser encontrado no GitHub através desse link curto:

http://bit.ly/mvc-livrocodigofonte

Utilize sempre que necessário para referência nos estudos e contribua com o código fonte, faça um fork e aguardamos seu pull request.

Além do código fonte, temos um grupo de discussão:

http://bit.ly/mvc-livrogrupodiscussao

Ele foi criado para conversarmos sobre ASP.NET MVC e dúvidas referentes ao livro. Aguardamos sua participação

Capítulo 2

Entendendo a estrutura de uma aplicação ASP.NET MVC

A web possui características intrínsecas, que não podem ser desconsideradas em hipótese alguma quando se planejam aplicações para serem executadas neste ambiente. Aspectos como: o correto e profundo entendimento do protocolo HTTP [4], utilização de tecnologias nativas dos navegadores (entenda-se, HTML, CSS e Javascript), responsividade, segurança, desempenho e, claro, *design*, devem estar **sempre** entre as principais preocupações de desenvolvedores web e, principalmente, das tecnologias e *frameworks* criados para gerar aplicações eficientes para este modelo.

ASP.NET MVC é uma tecnologia (*framework*) que foi concebida para respeitar tais premissas e mais, para facilitar (sim, facilitar) a vida dos desenvolvedores que a utilizam, uma vez que se pode escrever códigos mais limpos, livres de dependências e segmentados (separação de responsabilidades). Isso, por inércia, nos leva a uma estrutura de aplicação mais saudável, fácil de manter e otimizar.

2.1 ASP.NET MVC? Por quê?

Conforme mencionamos no capítulo 1, por muitos anos, a Microsoft apostou em um modelo diferente daquele tradicionalmente conhecido por desenvolvedores web (a saber, ASP.NET Webforms). Muito embora esta postura tenha colaborado em grande escala para arrebanhar desenvolvedores de outros universos para o da internet, isso acabou por gerar alguns "problemas" nas aplicações, dentre os quais citamos:

- **Dificuldade de manutenção**: como o nível de abstração (automatização de tarefas e encapsulamento de conceitos) no ASP.NET WebForms é elevado, as aplicações tendem a não utilizar boas práticas de desenvolvimento. Este fato, na grande maioria das vezes, implica em uma aplicação muito complexa de ser mantida;

- **Pouco controle na geração do HTML**: este é outro dos problemas gerados pela automatização dos processos. Quando se utiliza Visual Studio para criar aplicações ASP.NET clássicas, muito do HTML é gerado pela própria IDE (*Integrated Development Environment*) e pelos controles de servidor (GridView, ComboBox etc.). Isto prejudica o trabalho entre *designers* e desenvolvedores;

- **Difícil de testar**: eis aqui um grande problema encontrado nas aplicações ASP.NET tradicionais: são difíceis de testar. Note, com ASP.NET WebForms, basicamente o que fazemos em termos operacionais é arrastar um componente para a área de trabalho e adicionar um comportamento programático ao mesmo (via *code-behind*). Por que estamos mencionando isso? Para dizer que esta prática gera alto acoplamento, e isso inviabiliza, por exemplo, a realização de testes unitários. E aplicação sem teste nos dias atuais é inaceitável.

- **Preocupação com o ViewState**: cada elemento disponibilizado em uma aplicação ASP.NET trazia a possibilidade de se guardarem valores associados ao mesmo para evitar *gets/posts* desnecessários no servidor. Esta capacidade é conhecida no ASP.NET como *ViewState*. Uma das implicações da utilização não criteriosa deste recurso era o impacto negativo na performance das páginas;

- **Não separação de responsabilidades**: não era incomum em projetos ASP.NET WebForms encontrar páginas que acessavam bancos de dados, composição de elementos HTML através de strings em um método de exibição,

validações de CPFs no servidor etc. Separar responsabilidades era mais difícil e também gerava alto acoplamento.

Você sabe o que é "Alto acoplamento"?

Alto acoplamento é nome dado à ocorrência de alto grau de dependência entre partes de um software, sendo que estas partes deveriam funcionar de forma independente. Complicou?!

Pense na seguinte situação real, do dia a dia: Você precisa se conectar a um banco de dados para realizar uma atividade qualquer, certo? Para isso, provavelmente você possui um método que realiza a conexão com o banco. Acontece que, no interior de seu método de conexão, existe a chamada para outro método, sendo que este último apenas identifica se ocorreu algum erro e registra em um arquivo ou tabela de logs do sistema. Assim, sabemos que existe uma dependência explícita do método de conexão para com o método de registro de log. Um só pode funcionar se o outro funcionar.

Tal dependência chamamos de acoplamento. Se este fato ocorre de forma sistemática no projeto de software, dizemos que a aplicação é fortemente/altamente acoplada.

Vale ressaltar que apontar defeitos não é dizer que determinada tecnologia é ruim. O ASP.NET WebForms ainda é muito utilizado, mas — como tudo — tem prós e contras e, em alguns cenários, os defeitos apresentados por ele são maiores que os benefícios. Mesmo utilizando ASP.NET WebForms é possível ter um desenvolvimento que permite testes ou baixo acoplamento, mas devido à forma de trabalho com a tecnologia, o esforço para garantir esses ideais é alto utilizando ASP.NET WebForms.

Como o cenário web transformou-se rápida e amplamente nos anos posteriores (mudança alavancada principalmente pelos negócios, que também passaram a ser realizados neste ambiente), fez-se necessária a estruturação de uma nova tecnologia para desenvolvimento web dentro da plataforma .NET, que atendesse às novas demandas de mercado e que pudesse endereçar os problemas mencionados anteriormente. Surgiu então, o ASP.NET MVC.

2.2 Voltando à nossa primeira aplicação

No primeiro capítulo, criamos uma aplicação de exemplo (é claro que você se lembra, certo?!). Evidentemente, a aplicação à qual nos referimos é simples, afinal de contas, apresenta apenas uma mensagem personalizada na tela, não é mesmo?

Você deve se lembrar também que nossa aplicação exemplo foi criada baseada em um modelo (*template*) pré-construído, disponibilizado pelo Visual Studio. Esta escolha não foi por acaso. O *template* ao qual nos referimos traz uma estrutura de aplicação já pronta e segue algumas práticas interessantes, as quais estudaremos a seguir.

Olhando para a estrutura de diretórios

O primeiro aspecto para o qual gostaria de chamar a atenção na aplicação de exemplo é a estrutura de diretórios (apresentada pela figura 1.3).

A primeira importante consideração em relação à estrutura de diretórios apresentada é que, muito embora seja uma estrutura funcional e que atenda a boa parte das aplicações ASP.NET MVC, ela não é estática e/ou obrigatória. Sua aplicação pode possuir a estrutura de diretórios que achar mais conveniente. Entretanto, respeitando-se as convenções do *framework* MVC, você poderá encontrar alguns benefícios, os quais serão vistos de forma mais diluída nos capítulos posteriores.

Convenções?

Para automatizar algumas operações, o ASP.NET MVC utiliza algumas convenções (note que não se tratam de regras, mas de boas práticas a serem respeitadas). Seja para nomes de variáveis, objetos, classes e até mesmo diretórios.

Os diretórios mais importantes de nossa aplicação (justamente por estarem incorporados à convenção) são: *Models*, *Views*, *Controllers*, *App_Data* e *App_Start*. Suas respectivas funções são apresentadas a seguir:

- **Models**: responsável por agrupar os modelos de dados que serão utilizados pela aplicação. Você pode entender como "modelo de dados" tudo que se aplica à expressão, como por exemplo: arquivos EDMX (modelos do *Entity Framework*, XML's, webservices, entidades de negócio, DTO's [(]em português, "Objeto de Transferência de Dados"] etc.);

- **Views**: agrupa os elementos a serem visualizados pelo usuário final (as famosas *views*). Nome bem sugestivo, não?!
- **Controllers**: reúne as classes (e seus respectivos métodos) responsáveis por definir os comportamentos da aplicação a nível de servidor. Se facilitar, você pode entender este diretório como sendo o coração de sua aplicação web;
- **App_Data**: aqui o MVC entende que você poderá posicionar arquivos a serem consumidos por seu projeto, como imagens, vídeos, áudios etc;
- **App_Start**: arquivos que implementam comportamentos específicos e que devem ser inicializados junto com o projeto, devem ser posicionados neste diretório.

Existe ainda um diretório convencionado pelo ASP.NET MVC que não aparece em nosso projeto exemplo e que, portanto, não foi apresentado na lista anterior. Trata-se do *App_Code*. Ele não aparece na estrutura de diretórios da aplicação exemplo por um motivo muito simples: ele recebe arquivos Razor [3] provenientes de "Helpers" terceiros e a aplicação exemplo não implementa qualquer helper terceiro.

Falaremos muito de helpers durante os capítulos posteriores, por isso, já é importante que você se familiarize com este termo.

2.3 Um pouco mais sobre convenções

Convenção sobre Configuração, ou somente CoC, visa simplificar e diminuir as decisões que precisam ser tomadas pelo desenvolvedor. O trabalho fica muito mais fácil e simplificado já que algumas decisões estruturais e de nomes são sugeridos ou até exigidos em alguns casos, pelo *framework*.

Os *Controllers* e *Views* são ótimos exemplos.

No caso dos *Controllers* temos um diretório sugerido para a sua criação, e também temos que respeitar a convenção de que todo *controller* deve possuir o sufixo *Controller*, ou seja, `EmailController`, `ContatoController`, `ClientesController` etc.

As *Views* também possuem um diretório definido, e o MVC faz uso de uma convenção simples para identificar ou relacionar *Views* e *Controllers*. No caso do controller `EmailController`, uma pasta `Email` será criada dentro da pasta *Views*.

Podemos observar essas convenções na estrutura do nosso primeiro projeto. Veja na figura 2.1 os controllers `HomeController`, `AccountController` e as pastas `Home` e `Account` para as *Views*.

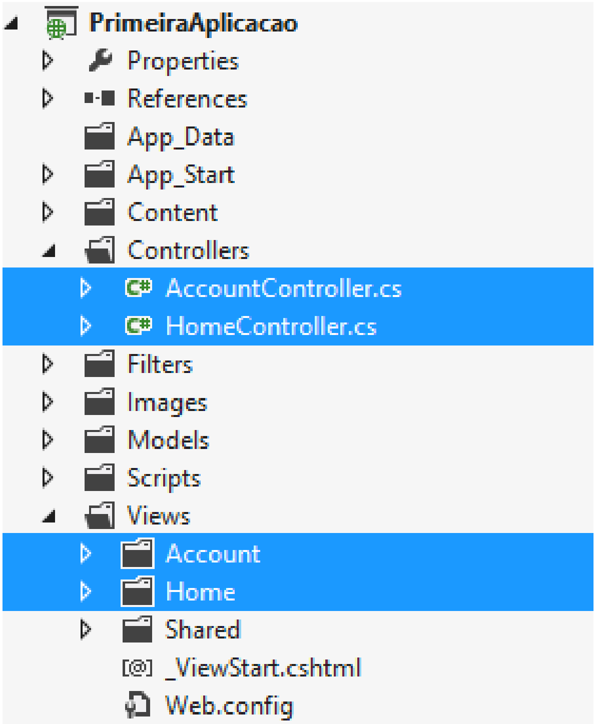


Figura 2.1: Convenção sobre Configuração no nosso primeiro projeto

Evidentemente que o *framework* MVC se responsabiliza por indexar os diretórios da solução (nome atribuído pelo Visual Studio para o ambiente que pode agrupar um ou múltiplos projetos), de forma com que seus respectivos conteúdos sejam facilmente acessíveis de qualquer ponto da aplicação.

Como você pôde perceber, em relação à estrutura de diretórios, temos um modelo bem definido e simplificado. Evidentemente, à medida que formos avançando, você verá este modelo sendo incrementado. Mas isto é assunto para os próximos capítulos.

2.4 Navegação baseada em rotas

A aplicação criada no primeiro capítulo apresenta outro aspecto fundamental relacionado ao modelo MVC de desenvolvimento: a **navegação baseada em rotas**. Volte suas atenções neste momento para a figura 1.4 do capítulo 1, mais precisamente, na barra de endereços exibida pelo navegador. Note que o endereço que está sendo exibido é http://localhost:1153/ . A pergunta que você pode estar se fazendo neste instante é: o que isso tem demais?

Para responder objetivamente a esta pergunta, convido-lhe a considerar a figura 2.2. Perceba que, mesmo adicionando novos parâmetros à URL, a exibição do conteúdo permanece inalterada.

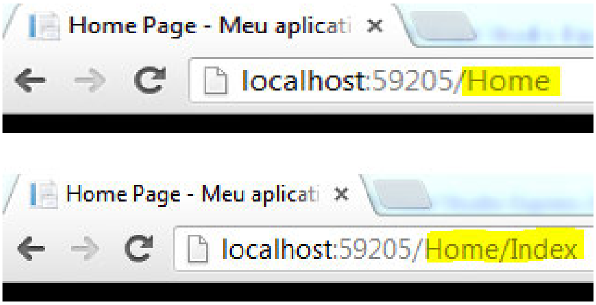


Figura 2.2: Adicionando parâmetros à URL

Não há nada de errado com este comportamento, muito pelo contrário. Na verdade, este é um dos grandes recursos implementados pelo *framework*. Graças a um mecanismo de roteamento de requisições convencionado, o ASP.NET MVC consegue entender que as três diferentes chamadas (http://localhost:1153/ , http://localhost:1153/Home e http://localhost:1153/Home/Index) estão na verdade fazendo referência ao mesmo recurso. Talvez as perguntas que permeiem sua mente neste instante são: onde está convencionado o mecanismo de rotas? Consigo visualizar e alterar essa configuração? A resposta direta e definitiva para esta pergunta é **sim** (veja a listagem 1).

Listagem 2.1 - Visualizando a configuração de rotas:

```
public static void RegisterRoutes(RouteCollection routes)
{
```

```
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home",
                        action = "Index",
                        id = UrlParameter.Optional
                      }
    );
}
```

Se você navegar até o diretório *App_Start* e expandi-lo através da *Solution Explorer* do projeto de exemplo, visualizará algumas classes escritas em C#. Todas elas implementam comportamentos que serão executados na inicialização da aplicação. Dentre estes arquivos, encontra-se um chamado `RouteConfig.cs`. Com um duplo clique sobre o mesmo você poderá visualizar o método apresentado pela listagem 1.

No código apresentado pela listagem anterior, ficam claros dois aspectos em relação ao mecanismo de rotas. O primeiro faz menção ao fato de que se pode registrar novas e personalizadas rotas (vide o nome do método `RegisterRoutes`) para uma aplicação. O segundo reside justamente no fato de que o ASP.NET MVC já traz consigo a implementação de uma rota padrão de navegação e é ela que gera, de fato, o comportamento apresentado pela 2.2.

Note que, para a rota `Default` (`name: "Default"`), temos a atribuição do valor `Home` para o atributo `Controller` (`controller = "Home"`), do valor `Index` para o atributo `Action` (`action = "Index"`) e do valor `UrlParameter.Optional` para o atributo `id` (`id = UrlParameter.Optional`).

Se ligarmos os pontos, concluiremos que a rota apresentada por nossa aplicação de exemplo (ver novamente figura 2.2) atende ao padrão definido pelo *framework*. Veja a comparação, padrão implementado:

```
controller = "Home", action = "Index", id = UrlParameter.Optional
```

Verifique a URL de nossa aplicação de exemplo:

http://localhost:1153/Home/Index

Ou simplesmente:

http://localhost:1153/

Vale lembrar mais uma vez que `Home` e `Index` são valores padrão na rota padrão do ASP.NET MVC. Opcionalmente, seria possível adicionar à URL um valor chave para identificar algum objeto. Desta forma, poderíamos ter algo como http://localhost:1153/Home/Index/200 por exemplo. Entretanto, muito embora seja possível realizar tal adição, no caso do exemplo discutido não faz sentido fazê-lo, pois não estamos identificando de forma unitária algum elemento (como http://localhost:1153/Clientes/Perfil/18556). Por este motivo, o último parâmetro da rota padrão é definido como opcional (`id = UrlParameter.Optional`).

A figura 2.3 apresenta um esquema funcional que ilustra (de forma simplificada) o mecanismo de roteamento implementado pelo ASP.NET MVC.

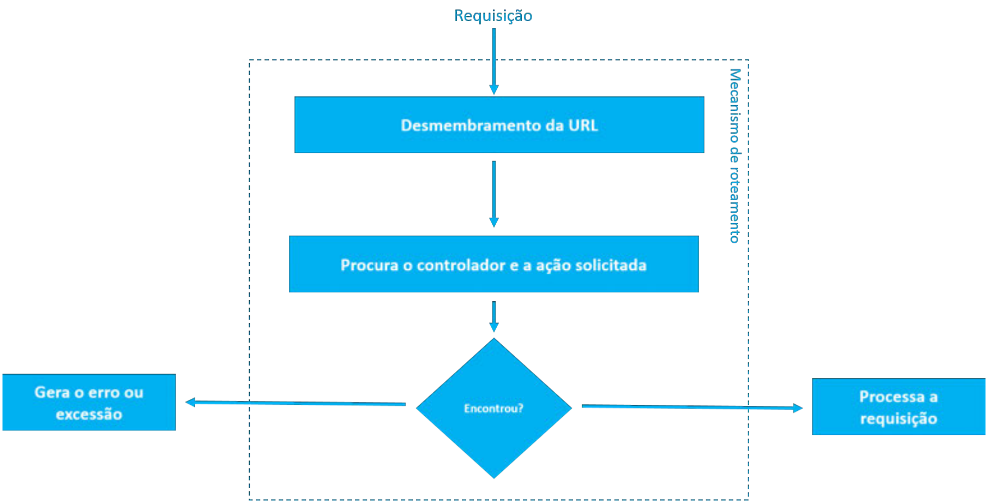


Figura 2.3: Esquema do mecanismo de rotas do ASP.NET MVC

Modelo de roteamento que gera "URLs semânticas"

Considere a seguinte URL: **http://seusite.com/Conteudo.aspx?id=987511**. A pergunta a ser realizada aqui é: é fácil memorizar uma URL como esta? A resposta, evidentemente, é não.

URLs como essa apresentada no parágrafo anterior são geradas automaticamente pelos sistemas web para fazer referência de forma dinâmica aos conteúdos da aplicação. Muito embora não haja qualquer problema técnico (isto é, esta URL funciona perfeitamente) relacionado a este tipo de URL, se pensarmos na experiência do

usuário final ao interagir com ela, poderemos encontrar alguns problemas, dentre os quais se destacam: dificuldade de memorização e baixa legibilidade.

Além disso, é importante observar o impacto negativo destas URLs nas técnicas de *Search Engine Optimization* (SEO) [2]. Note que os principais robôs de indexação de conteúdos de sites de busca (Google, Bing etc.), desconsideram URLs que apresentam alguns caracteres especiais, tais como: `&`, `@`, `%` etc.

Levando-se em consideração estes aspectos, ASP.NET MVC implementa através do mecanismo de rotas o modelo de URLs amigáveis. Assim, além tornar a URL da aplicação *indexável* para os robôs de busca, o *framework* melhora a experiência navegação do usuário, uma vez que é bem mais fácil de interagir com uma URL como **http://seusite.com/Conteudos/Artigos/987511**.

2.5 Concluindo e desafiando

Entender a estrutura de diretórios, as convenções utilizadas pelo *framework* MVC e o modelo de roteamento, é de fundamental importância para projetar aplicações ASP.NET MVC que usufruam do poder da plataforma em sua plenitude. No próximo capítulo, você será inserido no processo de planejamento de uma nova aplicação ASP.NET MVC. Esta aplicação será utilizada durante todo o livro para apresentar os demais conceitos e recursos da plataforma.

Neste ponto você já possui insumos suficientes para seguir através dos próximos capítulos. Nossa preocupação aqui fazer com que o conhecimento seja construído evolutivamente, pouco a pouco.

Desafio: Imagine que você precise adicionar um *plugin* jQuery para realizar alguma tarefa específica. Levando-se em consideração a convenção do ASP.NET MVC e a estrutura de diretórios proposta pela aplicação exemplo, adicione os arquivos do mesmo em seus respectivos lugares no projeto. Dica: `Content` não está lá por acaso.

Capítulo 3

Projetando a aplicação "Cadê meu médico?"

Nos primeiros capítulos deste livro, você foi apresentado aos conceitos iniciais relacionados ao *framework* ASP.NET MVC. Além disso, você criou sua primeira aplicação MVC que, muito embora tenha sido extremamente simples, esteve completamente funcional.

Apesar de básicos, os conceitos apresentados até aqui são de fundamental importância para o desenvolvimento de aplicações utilizando ASP.NET MVC. Daqui por diante, não só utilizaremos os conceitos já vistos, como iremos conhecer novos. Isso será realizado à medida que avançamos na criação de uma aplicação fictícia, chamada "Cadê meu médico?".

Nos próximos capítulos, avançaremos no desenvolvimento da aplicação "Cadê meu médico?". Você será guiado não apenas pelos detalhes da aplicação, como através de novos recursos do ASP.NET MVC. Apresentaremos também dicas e truques que podem facilitar em grande escala o trabalho ao utilizar *Models*, *Views* e *Control-*

lers.

Assim como a grande maioria das aplicações do mundo real, vamos construir passo a passo uma área administrativa para nossa aplicação, afinal de contas, precisamos saber quem está inserindo informações em nossa base de dados — e, claro, precisamos de segurança. Ao projetar a aplicação, levaremos em consideração também aspectos como a utilização em dispositivos móveis, boas práticas de programação e *design*.

3.1 Cadê meu médico?

Quem nunca precisou descobrir quais são os médicos de determinada especialidade disponíveis em sua cidade? "**Cadê meu médico?**" implementa justamente esta ideia. De forma rápida e fácil, os usuários poderão realizar consultas simples aos médicos disponíveis de forma segmentada por especialidade.

Para que a pesquisa funcione de forma satisfatória, médicos e especialidades deverão, evidentemente, ser cadastrados no sistema e em função disso, construiremos um *backend* com todos os cadastros necessários (o que já apresentará uma série de novos conceitos).

Assim, os seguintes recursos serão implementados:

- **Área administrativa da aplicação**: área de acesso restrito, onde os usuários precisarão realizar o processo de autenticação para ter o devido acesso. Além do mecanismo de *login*, dentro da área administrativa serão implementados todos os cadastros (médicos, especialidades, cidades e usuários);

- **Gerenciamento de médicos**: acoplada à área administrativa criaremos uma subárea para CRUD (*Create*, *Read*, *Update* e *Delete*) de novos médicos;

- **Gerenciamento de especialidades médicas**: também acoplada à área administrativa, será criada uma subárea para CRUD de especialidades médicas;

- **Página pública para consulta de médicos por especialidade**: criaremos uma página pública, onde disponibilizaremos os recursos necessários para que a consulta de médicos possa ser realizada;

- **Versão do site para dispositivos móveis**: conforme mencionado anteriormente, temos a preocupação de que nossa aplicação possua uma experiência adequada para dispositivos móveis. Desta forma, criaremos uma nova página

(também pública) de consulta aos médicos mas, desta vez, voltada para dispositivos móveis.

Para que a proposta da aplicação "Cadê meu médico?" torne-se mais clara sob os pontos de vista funcional e arquitetural, apresentamos alguns detalhes acerca da mesma. Assim, convidamos a considerar a figura 3.1.



Figura 3.1: Visão geral da aplicação 'Cadê meu médico?'

Além da visualização da estrutura, neste momento, um esforço adicional se faz necessário. Trata-se da construção e posterior apresentação de um diagrama de entidades e seus respectivos relacionamentos. Evidentemente, ao realizarmos tal operação, já estamos pensando na estrutura final de banco de dados de nossa aplicação.

A figura 3.2 apresenta o DER (Diagrama Entidade-Relacionamento) proposto.Para chegarmos a este modelo, estamos considerando algumas regras de negócio para o sistema. São as principais:

- Um médico deve possuir uma especialidade médica;
- Uma especialidade médica pode estar associada a diferentes médicos;
- Um médico poderá estar associado apenas a uma cidade;
- Uma cidade pode possuir diversos médicos;
- O sistema deverá controlar o acesso de usuários ao sistema administrativo.

Diagrama Entidade-Relacionamento?

Diagrama Entidade-Relacionamento (ou simplesmente DER) é uma metodologia que permite criar, e posteriormente exibir, de forma gráfica e simplificada — nível de abstração mais alto — uma estrutura mais complexa de regras e agrupamento de dados em uma aplicação.

Através da representação gráfica simples possibilitada pelos DER's, é possível entender a lógica de dados e, até mesmo, algum comportamento da aplicação. Além disso, os diagramas expressam os relacionamentos entre as entidades importantes para o sistema, o que facilita o entendimento da situação problema e posterior implementação, tanto para desenvolvedores quanto para administradores de bancos de dados.

Resumindo: Trata-se do "mapa" gerado pelos analistas de requisitos e passado para os desenvolvedores, arquitetos de software e administradores de bancos de dados.

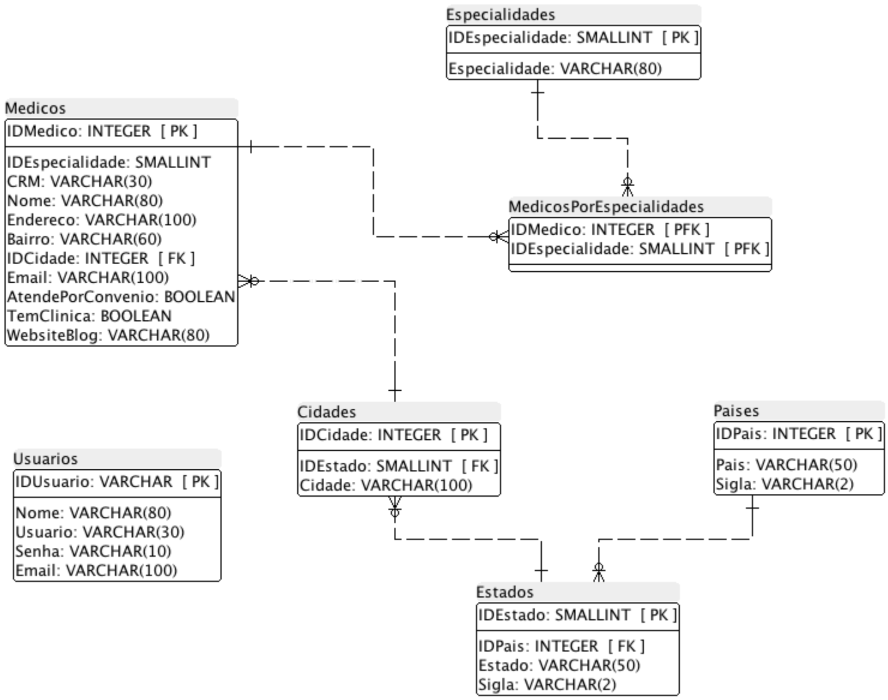


Figura 3.2: DER para a aplicação ‘Cadê meu médico?’

São legendas para a figura 3.2:

- [PK]: é o acrônimo para *Primary Key*. Em português, “Chave Primária";
- [FK]: é o acrônimo para *Foreign Key*. Em português, “Chave Estrangeira";
- [PFK]: é o acrônimo para *Primary Foreign Key*. Em português, “Chave Primária e Estrangeira";
- Na sequência, você pode visualizar as entidades participantes do modelo (Médicos, Especialidades etc.)

3.2 Criando a aplicação

Agora já sabemos tudo o que precisamos, portanto, é hora de começar a criar efetivamente a aplicação **Cadê meu médico?**. Com o Visual Studio em execução, navegue

até o menu superior e lá selecione a opção `File > New > Project`. Na janela que se apresentará, selecione à esquerda a opção "web" e na sequência, a opção "ASP.NET MVC Application" à direita. A figura 3.3 apresenta a seleção a qual nos referimos.

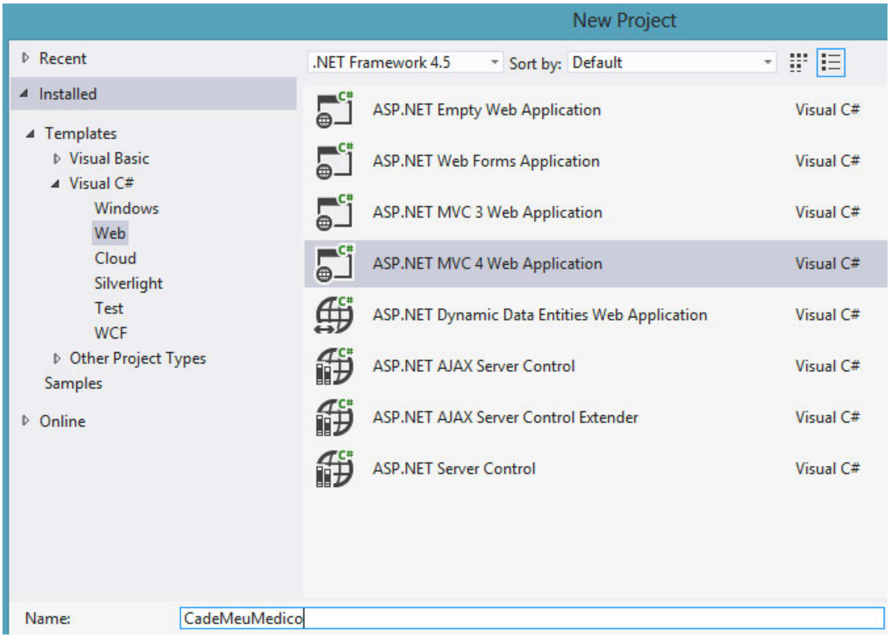


Figura 3.3: Selecionando o tipo de projeto ASP.NET

Isto feito, uma nova janela será apresentada pelo Visual Studio. Ela exibe alguns *templates* disponíveis para a criação de uma aplicação ASP.NET MVC. Você deve se lembrar que em nossa primeira aplicação de exemplo, escolhemos o *template* "*Internet Application*". Como resultado desta escolha, o Visual Studio cria uma estrutura base e adiciona várias referências ao projeto, além de definir um layout padrão.

Para este segundo e mais completo exemplo, vamos utilizar o *template blank*, através do qual o Visual Studio cria apenas a estrutura básica de diretórios, e adiciona os códigos de inicialização da aplicação (como a configuração do roteamento padrão, por exemplo). A figura 3.4 mostra a seleção mencionada.

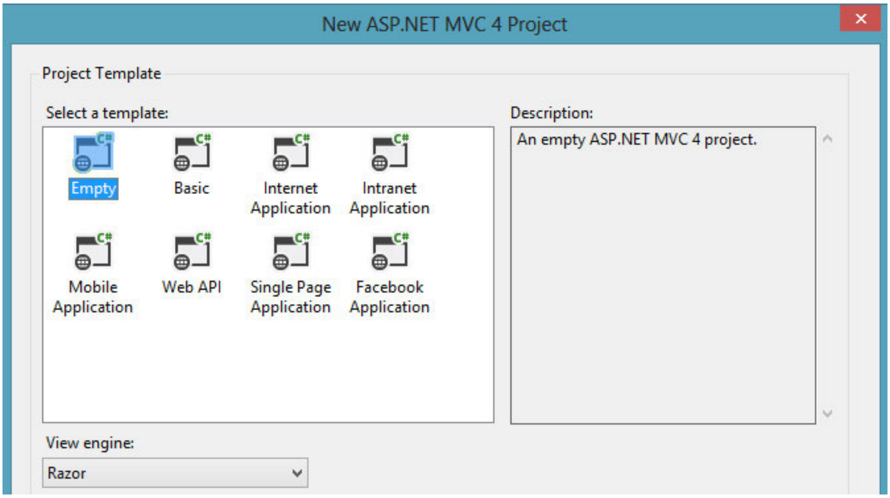


Figura 3.4: Selecionando o template do projeto

Utilizando o *template* básico, temos algum trabalho para criar o *layout*, adicionar referências etc. Muito embora estejamos conscientes em relação a este fato, isto está sendo feito propositalmente, uma vez que servirá para apresentar truques e exaltar ferramentas do Visual Studio que nos auxiliam na construção das aplicações. Vale lembrar que quando começamos um projeto, nem sempre podemos utilizar o *layout* e bibliotecas que a ferramenta sugere. Por isso o conhecimento para criar uma aplicação desde o início é importante!

Após a criação do projeto, a estrutura de diretórios apresentada no *Solution Explorer* deverá ser semelhante à apresentada pela figura 3.5.

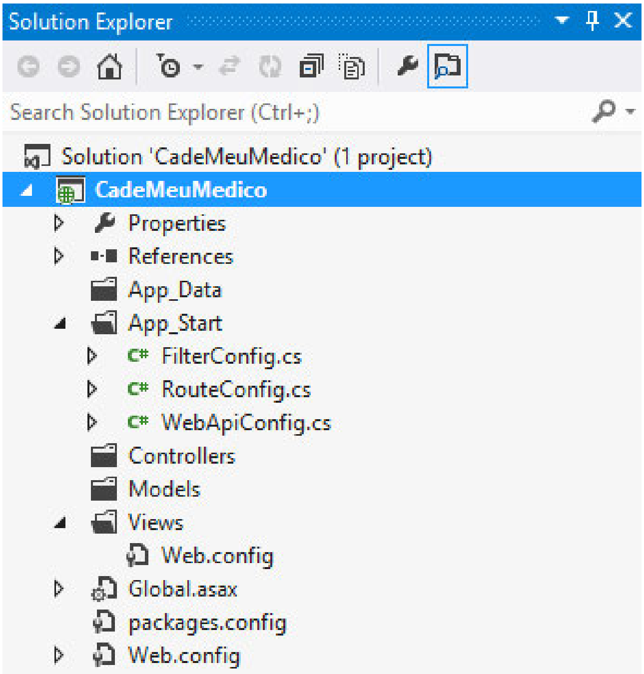


Figura 3.5: Estrutura inicial da aplicação Cadê meu Médico?

3.3 Referenciando bibliotecas

No desenvolvimento de qualquer aplicação, web ou não, é comum a utilização de bibliotecas já prontas que encapsulam tarefas específicas e que auxiliam no desenvolvimento. Entre os benefícios proporcionados por esta prática, podemos citar a produtividade e o reaproveitamento de código. Com aplicações ASP.NET MVC não é diferente e, por isso, você verá ao longo da construção deste projeto, a adição de várias referências que facilitarão em grande escala a realização de determinadas tarefas.

Quando criamos nossa primeira aplicação de exemplo (ver capítulo 1) e escolhemos o *template* "*Internet Application*", vimos o Visual Studio criar um *layout* padrão para a aplicação e também adicionar, de forma automática, referências a várias bibliotecas javascript. Além disso, bibliotecas para que alguns recursos nativos do *template* padrão pudessem funcionar de forma adequada (autenticação de usuários, por exemplo) foram adicionadas. Mencionamos isso apenas para esclarecer que trabalhar com bibliotecas em projetos ASP.NET MVC é normal e você deve, sempre que

necessário, se valer desta opção.

Como estamos optando pela utilização do *template* "*Empty*" para criar a aplicação **Cadê meu médico?**, precisaremos adicionar manualmente todas as referências às bibliotecas necessárias. Além disso, não temos um *layout* pré-construído, o que nos forçará a criar o nosso próprio.

Para que possamos criar um bom nível de interatividade entre o usuário final e a aplicação **Cadê meu médico?**, utilizaremos um conhecido *framework* javascript, a saber, jQuery. Para nos auxiliar na criação do *layout* da aplicação, utilizaremos um excelente componente de elementos gráficos que é amplamente difundido e utilizado em projetos web e que é disponibilizado gratuitamente pelo time de designers do Twitter — o **Bootstrap**.

- **jQuery**: é uma das mais famosas e funcionais bibliotecas baseadas em javascript do mundo. Facilita o trabalho e manipulação dos objetos DOM (*Document Object Model*), chamadas Ajax, manipulação de eventos e animações;

- **Twitter Bootstrap**: trata-se de uma biblioteca de CSS (*Cascading Style-Sheet*) e componentes jQuery que facilita o trabalho de estruturar ou criar novos *layouts* para aplicações;

Encontrar e utilizar bibliotecas úteis nos projetos pode ser uma tarefa complexa, tendo em vista a enorme "oferta" de bibliotecas para solucionar os mesmos tipos e classes de problemas. Assim, é preciso estar atento para escolher aquela que melhor atende às necessidades do projeto. Além disso, existem outros aspectos que acabam gerando problemas no gerenciamento de bibliotecas: instalação, atualização, dependências etc.

A boa notícia é que os problemas mencionados anteriormente não existem mais se você é usuário do Visual Studio. Isso graças a uma ferramenta introduzida pela Microsoft no Visual Studio 2010 com suporte ao ASP.NET MVC a partir da versão 3, chamada *NuGet*. A lista a seguir apresenta alguns dos benefícios proporcionados pelo *NuGet*.

- **Repositório de bibliotecas**: o NuGet possui um repositório oficial onde qualquer biblioteca pode ser cadastrada e disponibilizada. Toda pesquisa é realizada nesse repositório público;

- **Ranking**: quando uma pesquisa é realizada, as bibliotecas são ordenadas pelo número de *downloads* realizados. Logo, os mais baixados aparecem no topo da lista;

- **Instalação**: as responsabilidades de *download* e instalação são do *NuGet*. Como usuários, precisamos informar apenas qual o projeto no qual a biblioteca deve ser instalada;
- **Atualização**: quando uma biblioteca é atualizada no repositório oficial do *NuGet*, ela entra na lista de bibliotecas que podem ser atualizadas no seu projeto. Isso é muito útil já que, para a maioria das bibliotecas, as versões com correções e melhorias de desempenho são liberadas frequentemente;
- **Dependências**: imagine que você precisa da biblioteca *A*, mas para funcionar ela depende da *B* — a automatização do *download* e instalação das dependências também é feita automaticamente pelo *NuGet*.

A princípio, o *NuGet* nos ajudará nos processos de instalação do jQuery e do Twitter Bootstrap. Para isso, navegue até o menu `Tools > Library Package Manager > Manage NuGet Packages for Solution`. Na janela que se apresentará, selecione do lado esquerdo a opção `OnLine:> NuGet official package source`, e no canto superior direito pesquise por *jQuery*. No centro da janela serão apresentados os resultados da pesquisa. No item *jQuery* clique no botão *Install*. A figura 3.6 ilustra este processo.

Ao clicar no botão *Install*, uma janela será apresentada com os projetos da nossa solução. Escolha o projeto no qual a biblioteca deverá ser instalada, conforme ilustra a figura 3.7.

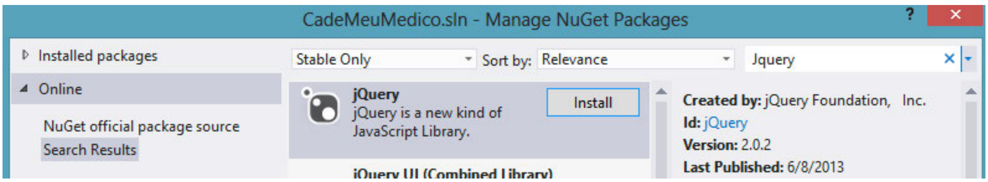


Figura 3.6: Pesquisando e instalando jQuery com o NuGet

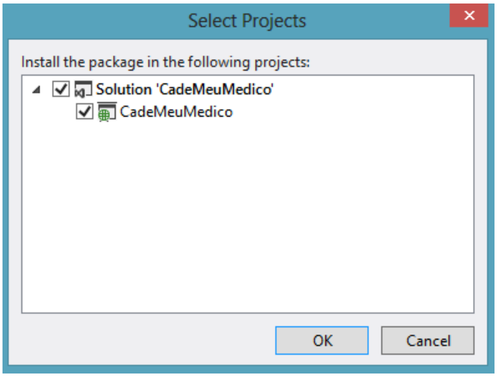


Figura 3.7: Definindo em qual projeto o NuGet irá instalar a biblioteca

Agora que já temos o *jQuery* devidamente instalado em nosso projeto, realizaremos o mesmo procedimento para o Twitter Bootstrap. Pesquise por "Bootstrap" e na sequência, efetue sua instalação. A figura 3.8 ilustra este processo. Estamos utilizando a versão 3.0.0.0 do Bootstrap, que é a última versão lançada até a publicação desse livro. Caso sua pesquisa retorne uma versão superior, fique atento ao site oficial por possível guias de migração de uma versão para outra.

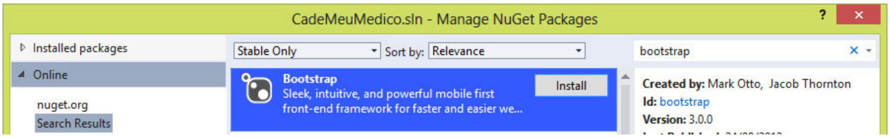


Figura 3.8: Pesquisando e instalando Twitter Bootstrap com o NuGet

É pertinente observar neste ponto o fato de que, para instalar as bibliotecas necessárias ao nosso projeto via NuGet, estamos utilizando a ferramenta gráfica disponibilizada pelo Visual Studio para este fim. Muito embora tal ferramenta seja funcional e atenda às necessidades, vale mencionar que esta não é a única forma de trabalho disponível. Desenvolvedores que possuem preferência por trabalhar com linhas de comando também são atendidos pelo Visual Studio. Isso é feito através da ferramenta *Package Manager Console*, acessível através do menu superior na opção *Library Package Manager*.

Ao acessar a opção mencionada, o Visual Studio apresentará um *console* no qual os comandos poderão ser digitados. Você poderá fazer praticamente qualquer ope-

ração com NuGet via linha de comandos. A "linguagem" aceita pelo NuGet para os *inputs* de comandos é o *PowerShell*. Você pode encontrar um guia de referência completo sobre os comandos aceitos pelo NuGet via *PowerShell* seguindo o link:

http://bit.ly/referenciapowershell

Para saber mais sobre o *PowerShell*, você deve pode utilizar:

http://bit.ly/guiapowershell

3.4 Criando o *layout* da aplicação

Nosso projeto já se encontra criado e suas respectivas bibliotecas iniciais (indispensáveis para iniciar a construção de nossa aplicação) já estão referenciadas. Estamos aptos, portanto, a avançar em nosso projeto. Iniciaremos pela parte visual pois através dela tiraremos lições preciosas acerca do *framework* ASP.NET MVC.

Os itens anteriores deste capítulo apresentaram a criação do projeto e o modelo de referenciamento via *NuGet*. Como próxima etapa, precisamos testar se a referência está funcionando de forma adequada. Para isso, vamos criar um *layout* inicialmente de teste e claro, na sequência, iremos incrementá-lo. Assim, além de validarmos o funcionamento das bibliotecas, definimos também a estrutura básica de toda a aplicação.

Muito embora tenhamos no futuro um capítulo dedicado especificamente para tratar do assunto "*views*", construiremos agora algumas delas, que servirão de base para nosso aplicativo. É importante notar que o objetivo não é apresentar detalhes sobre *views*, mas sim, sobre o processo de estruturação de um *layout* utilizando ASP.NET MVC.

Também utilizaremos algumas funções do ASP.NET *Razor*, que é o motor de renderização das páginas criadas com o *ASP.NET MVC*. Mas não se preocupe com isso neste momento. No capítulo 9 você será guiado também pelos detalhes acerca do *Razor*.

Um dos muitos conceitos interessantes introduzidos pelo *ASP.NET* foi o de *Master Pages* (páginas mestras). Através das famosas *master pages*, conseguimos reaproveitar muito do *layout* que é comum ao site. Por exemplo, não precisamos construir o menu em cada página do site. Ao invés disso, podemos colocar o menu em uma *:master page* e "dizer" ao ASP.NET que todas as páginas que herdam os estilos de determinada página mestra devem exibir o mesmo menu. De igual forma, esta técnica poderia ser utilizada para estruturar quaisquer conteúdos de propósitos gerais, permitindo assim a manutenção do foco no conteúdo específico da página corrente.

O modelo de trabalho com *master pages* oferecido pelo ASP.NET MVC é ligeiramente diferente daquele proporcionado pelos *web forms*. Você pode encontrar um bom texto de referência acerca da utilização de *master pages* com *web forms* seguindo o link apresentado pela referência [5].

Muito embora as abordagens sejam diferentes, o conceito é rigorosamente o mesmo e iremos utilizá-lo amplamente em nosso projeto. Assim, na *Solution Explorer*, clique com o botão direito do mouse na pasta *Views* e navegue pelas opções `Add > View`. A figura 3.9 ilustra este processo.

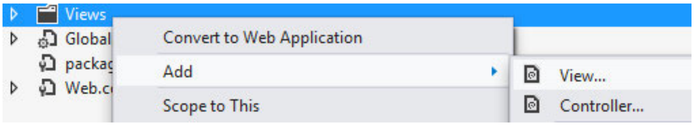


Figura 3.9: Criando uma nova View

Na sequência, uma janela onde iremos informar mais detalhes acerca de nossa *View* será apresentada. Informe o nome *Layout*, e desmarque a opção *Use a layout or master page*. Essa opção é justamente a que vamos marcar na criação de alguma View que irá utilizar nossa Master Page. Feito isso, clique no botão *Ok*. A figura 3.10 mostra essa etapa.

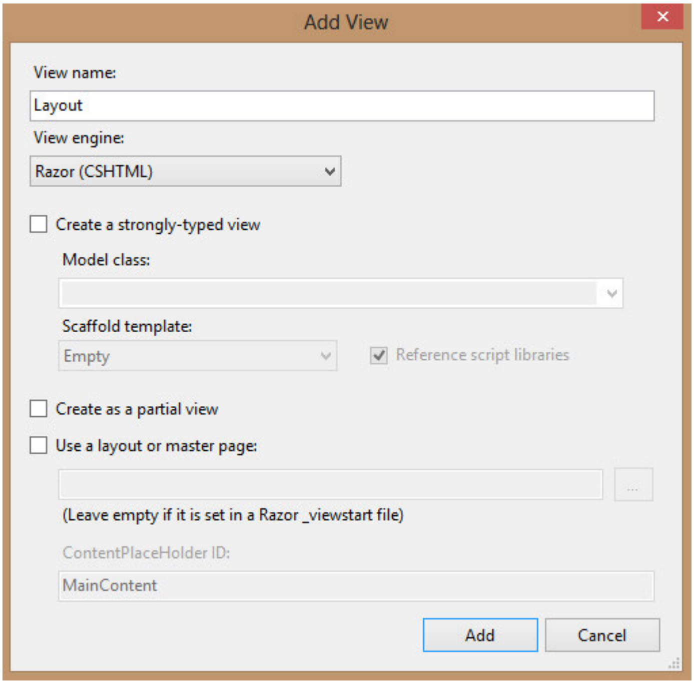


Figura 3.10: Criando uma nova View

Após a criação do arquivo, o Visual Studio vai abrir a nossa View para edição. Você deverá ver um código parecido com o código da listagem 1.

Listagem 3.1 - Código inicial da master page:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Layout2</title>
</head>
<body>
    <div>
```

```
    </div>
</body>
</html>
```

Antes de modificar o HTML gerado pelo Visual Studio para a *master page*, é preciso notar um aspecto importante. Note que, no início do código, à propriedade "*Layout*" é atribuído o valor "*null*". Pois é justamente esta a propriedade que recebe a indicação do *layout* padrão, ou seja, a *master page* da qual a página herdará.

Precisamos concordar, neste ponto, que executar tal atribuição em todas as páginas de uma aplicação não é lá muito produtivo. Imagine um projeto grande, com muitas *views*. Esta prática tornaria onerosas demais tanto a implementação quanto a manutenção dos arquivos de *layout* do projeto no caso de uma possível troca de *master page*. Além disso, salta à vista a ausência do conceito de reutilização de código.

O que faremos agora é criar uma nova *view* que será responsável por definir essa propriedade em todas as *views* da nossa aplicação. Para isso, novamente vá até a *Solution Explorer*, clique com o botão direito do mouse na pasta *Views* e navegue pelas opções `Add > View`. A figura 3.11 ilustra este processo.

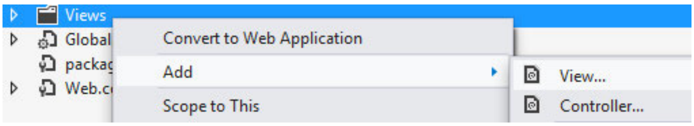


Figura 3.11: Criando uma nova View

Agora, na janela onde informaremos os detalhes acerca de nossa nova *view*, utilizaremos o nome `_ViewStart`. Ainda nesta janela, desmarque também a opção *Use a layout or Master Page*, e clique em *Ok*.

Note que, novamente, a *Convenção sobre Configuração* nos ajuda neste caso. Por quê? Uma *view* chamada `_ViewStart` será executada no início da renderização de todas as *views* da aplicação. Voltando à configuração, vamos configurar a propriedade *Layout* na nossa `_ViewStart`, conforme apresenta a listagem 2.

Listagem 3.2 - Propriedade que define a master page:

```
@{
```

```
    Layout = "~/Views/Layout.cshtml";
}
```

Podemos voltar agora e definir a estrutura padrão de *layout* da aplicação no arquivo "Layout.cshtml". Já utilizamos um pouco de *Razor* na criação da `_ViewStart` e agora iremos avançar um pouco mais. A listagem 3 apresenta o *layout* padrão da nossa aplicação.

Listagem 3.3 - Layout padrão da aplicação:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Cade meu medico? - @ViewBag.Title</title>
    <link href="@Url.Content("~/Content/bootstrap/bootstrap.min.css")"
    rel="stylesheet" />
    <style>
        body {
            padding-top: 60px;
        }
    </style>
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <a class="navbar-brand" href="#">Cade meu medico?</a>
            </div>
            <div class="collapse navbar-collapse">
                <ul class="nav navbar-nav">
                    <li class="active"><a href="#">Home</a></li>
                    <li id="menuPaises">
                        @Html.ActionLink("Países", "Index", "Paises")
                    </li>
                </ul>
            </div>
        </div>
    </div>

    <div class="container">
```

```
        @RenderBody()
    </div>

    <script src="@Url.Content("~/Scripts/jquery-2.0.2.min.js")">
    </script>
    <script src="@Url.Content("~/Scripts/bootstrap.min.js")">
    </script>
    @RenderSection("script", required: false);

</body>
</html>
```

Além de básico, o HTML de nossa estrutura contém a referência às bibliotecas de CSS e JavaScript que utilizaremos na aplicação. Além disso, você deve ter notado que em alguns pontos utilizamos o caractere *"@"*. Pois saiba que é exatamente nestes pontos que estamos utilizando o ASP.NET *Razor*.

Outro aspecto a ser notado reside na definição do título das páginas de nossa aplicação. Estamos concatenando o texto “Cadê meu médico?” com `"@ViewBag.Title"`. Como você deve se lembrar, quando personalizamos a mensagem da nossa primeira aplicação, utilizamos a propriedade `ViewBag` para enviar nossa mensagem para a *view*. Agora estamos utilizando novamente a `ViewBag`, mas para mostrar nas páginas os valores predefinidos.

O *Razor* possui uma grande quantidade de métodos que auxiliam os *desenvolvedores* em grande escala no processo de construção de aplicações. Um deles é o *"@Url.Content()"*, que converte um diretório virtual relativo para um diretório absoluto na aplicação. Através disso, podemos vincular nossos arquivos .css e .js (por exemplo) sem nos preocuparmos com a estrutura de publicação da aplicação no IIS (*Internet Information Services*).

Lembre-se que todas as paginas irão utilizar o *layout* padrão, portanto, não precisamos repetir alguns códigos comuns, mas é preciso definir em nosso *layout* onde o código das páginas será renderizado em nosso *layout* padrão. É isso que estamos fazendo ao utilizarmos o método *"@RenderBody()"* do ASP.NET *Razor*. Quando uma *view* for renderizada, seu código será adicionado naquele ponto da página HTML.

Um outro truque muito interessante do *Razor* são as *Sections*. Através delas, podemos definir seções onde determinados códigos serão renderizados. As futuras *views* de nossa aplicação possuirão código JavaScript próprio, e eles precisam ser renderizados após a definição das bibliotecas JavaScript que estamos utilizando. Desta forma, com o `@RenderSection("script", required:false)`, criamos uma

seção onde os scripts serão renderizados. Como nem toda página possuirá algum *script*, essa seção não é obrigatória e por isso, utilizamos o *"required:false"*.

Layout padrão criado, é hora então de testar o mesmo com uma página de teste em nossa aplicação. Na *Solution Explorer*, clique com o botão direito na pasta *Controllers* e navegue até as opções `Add > Controller`. A figura 3.12 ilustra este processo.

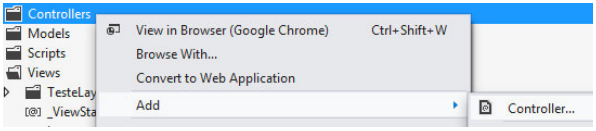


Figura 3.12: Adicionando um novo Controller

Na janela que será exibida a seguir, defina o nome do *controller* como "TesteLayoutController". Lembre-se, todo controller deve possuir o sufixo *Controller* (mais uma vez o CoC atuando). A figura 3.13 representa isso.

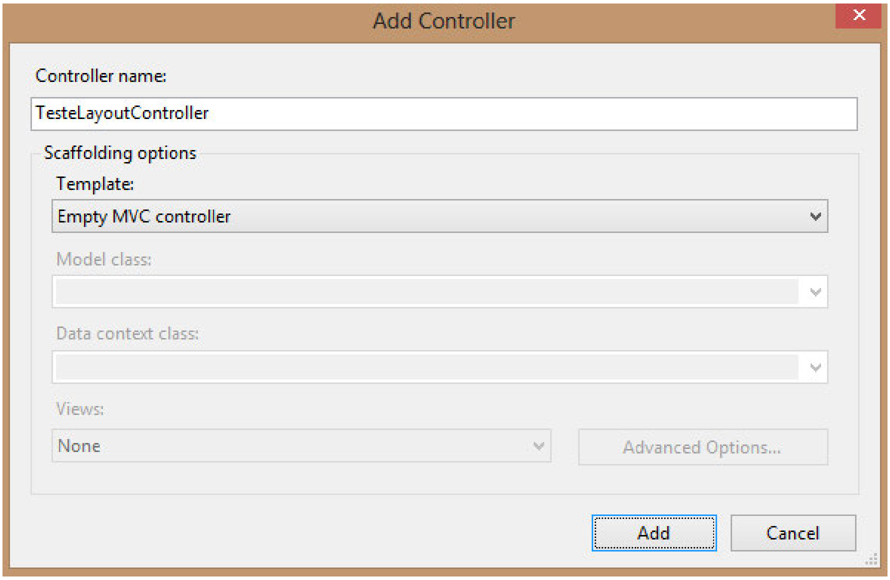


Figura 3.13: Adicionando um novo Controller

Após executarmos a criação do *controller*, o Visual Studio abrirá o código do

arquivo. Evidentemente, ele deverá se parecer com o código da listagem 4.

Listagem 3.4 - Controller do teste do layout:

```
public class TesteLayoutController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

O próximo passo consistirá da criação de uma nova *view*, que será apresentada quando o método *Action* do *Controller* for chamado. Recorde que uma das Convenções sobre Configuração é que a *view* deverá ficar em uma pasta com o nome do *Controller*, e a *View* deverá ter o nome da *Action*. Felizmente o Visual Studio automatiza esse trabalho!

Mais uma vez é importante observar que você não deve estar tão preocupado se estiver achando estranho os termos *Controller*, *Action* etc. Eles serão apresentados de forma detalhada nos capítulos a seguir.

Clique com o botão direito do mouse sobre o código `View()` e navegue até a opção *Add View*. A figura 3.14 ilustra este processo.

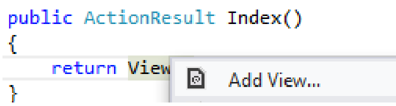


Figura 3.14: Adicionando uma nova View

Na janela que será apresentada, você poderá notar que o nome da *view* já estará definido, e se a opção *"Use a layout or master page"* não estiver selecionada, você deve fazê-lo. Não informe nenhum valor no campo. Com o campo em branco, o MVC vai usar por padrão o arquivo criado anteriormente (a saber, `_ViewStart`). A figura 3.15 mostra essa etapa.

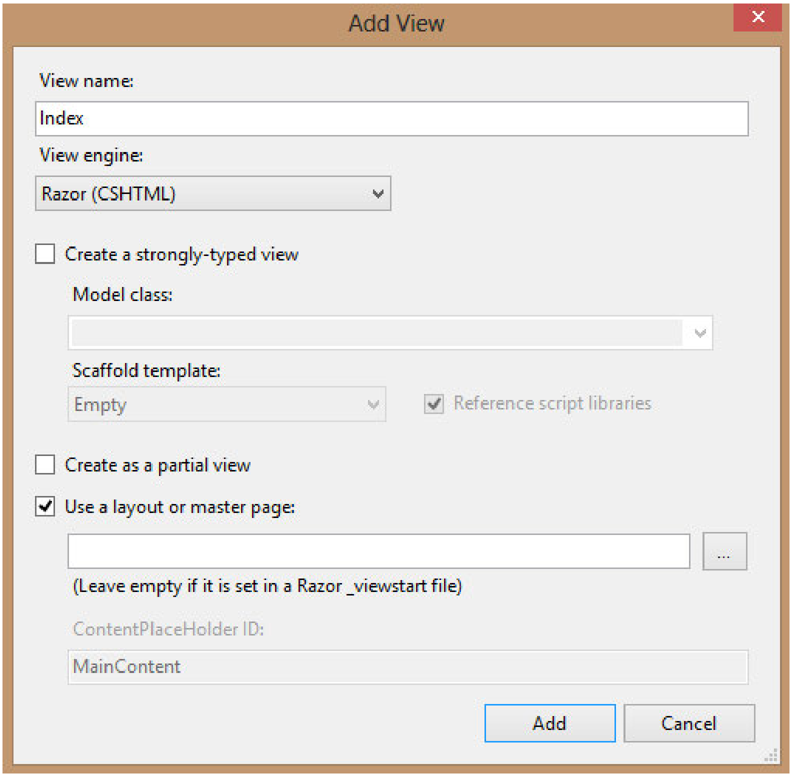


Figura 3.15: Adicionando uma nova View

Após a criação do arquivo, o Visual Studio abrirá o código e ele deve estar parecido com a listagem 5. Repare que podemos alterar na própria *view* o valor do `ViewBag.Title`, que será concatenado com o título das páginas. Altere o valor para *"Teste de layout"*

Listagem 3.5 - Código da view de teste do layout:

```
@{
    ViewBag.Title = "Teste de layout";
}

<h2>Index</h2>
```

Feito isso, compile seu projeto, através do pressionar da tecla `F5` no Visual Studio. Quando o aplicativo for carregado, provavelmente uma página de erro será mos-

trada, já que não definimos um Controller padrão. Mas tudo bem, isto é proposital. Digite ao final da URL a fração de `string` "/testelayout" que nossa página de teste será exibida. A figura 3.16 ilustra este processo.

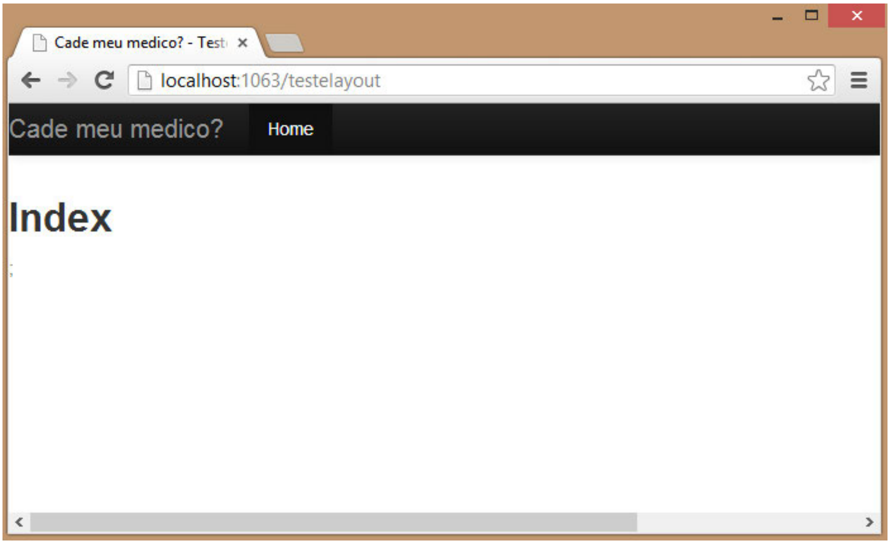


Figura 3.16: Executando a página com o teste do layout

3.5 Concluindo e desafiando

O objetivo deste capítulo foi apresentar a você algumas das principais opções para se estruturar *layouts* de aplicações web utilizando o ASP.NET MVC com a *engine* Razor. Você aprendeu também que, muito embora o processo de utilização das famosas *master pages* seja diferente na *framework* MVC, ele está presente e é amplamente utilizado no processo de construção de aplicações web.

No teste anterior, ao carregar o aplicativo um erro foi apresentao, pela falta do Controller padrão. Para resolver esse pequeno problema precisamos criar um novo controller com uma view que represente `/home/index`. Com o que você aprendeu aqui, crie esse *controller* e *view* já utilizando a master page do site.

Capítulo 4

Models: Desenhando os modelos da nossa aplicação

Até aqui você foi apresentado a uma série de importantes conceitos e recursos relacionados ao desenvolvimento de aplicações com ASP.NET MVC. Já falamos sobre convenções (CoC) da *framework*, sobre o modelo de roteamento, *master pages*, referenciamento de bibliotecas via *NuGet*, dentre outros. Estes conceitos serão de fundamental importância para executar as operações que faremos daqui até o final do livro.

No capítulo 3, tivemos a oportunidade de visualizar os aspectos gerais (estruturais, inclusive) da aplicação de exemplo que norteará todo o desenrolar dos fatos daqui por diante. Um dos aspectos apresentados relacionado a esta aplicação no capítulo anterior foi o DER (ver figura 3.2). Conforme mencionamos naquele momento, o DER traz toda informação de que precisaremos para criar a base de dados de nossa aplicação.

Neste capítulo, como o próprio nome sugere, iremos detalhar um pouco mais

o assunto *models*, ou simplesmente, modelo, já que ele é parte integrante da sigla **M**VC e constitui parte importante das aplicações constituídas neste formato. Tendo isto em mente, estamos prontos para prosseguir!

4.1 Model?!

Um dos grandes objetivos que o *framework* ASP.NET MVC pretende alcançar é a separação de responsabilidades entre as camadas de uma aplicação. Neste contexto, a palavra "separação" poderia tranquilamente ser substituída pela palavra "isolamento", uma vez que esta última passa mais o sentido que pretende endereçar.

Como já sabemos, os *models* são a parte de um projeto ASP.NET MVC responsável por agrupar — tendo em mente este modelo separatista do *framework* — as fontes de dados de uma aplicação, seja esta fonte de dados for um banco de dados, um *web-service*, um arquivo XML, um arquivo texto, entre outros. A ideia fundamental é que as fontes de dados fiquem isoladas de forma literal das demais camadas.

Mas para que possamos entender os *modos operandi* do *framework* junto aos *models*, apresentaremos algumas abordagens possíveis de forma atrelada ao nosso projeto de exemplo. Mãos à obra?!

4.2 Abordagens para a criação de *Models*

Em termos práticos, quando o assunto é "conexão com fontes de dados" em uma aplicação ASP.NET MVC, estamos falando necessariamente em conexão através de *models* a fontes de dados heterogêneas (serviços, arquivos etc.), conforme havíamos mencionado no início deste capítulo. Se mudarmos esta expressão para "conexão com bancos de dados" (que é o que faremos de fato em nossa aplicação exemplo) em uma aplicação ASP.NET MVC, ainda assim, estaremos falando de conexão através de *models*, entretanto, é de fundamental importância entender que para esta abordagem, existem diferentes formatos para se realizar tal procedimento. Os três mais difundidos no mercado são:

- Através da utilização direta das classes ADO.NET;
- Através de ORM's (no modelo *database first*);
- Através de ORM's (no modelo *code-first*);
- Através de ORM's (no modelo *model-first*).

Uma pitadinha de ADO.NET

ADO é o acrônimo para *ActiveX Data Object*. Trata-se de um conjunto de classes e recursos pertencentes ao conjunto de *namespaces* da plataforma .NET (daí o nome ADO.NET), disponibilizado para possibilitar o acesso às estruturas de bancos de dados por parte das aplicações. Através dela, é possível realizar diferentes tipos de acessos. Você pode optar, por exemplo, entre buscar dados na memória do servidor ou ir até o banco de dados a cada requisição, dentre diversos outros recursos. A figura 4.1 apresenta uma visão conceitual geral do *namespace* **System.Data**.

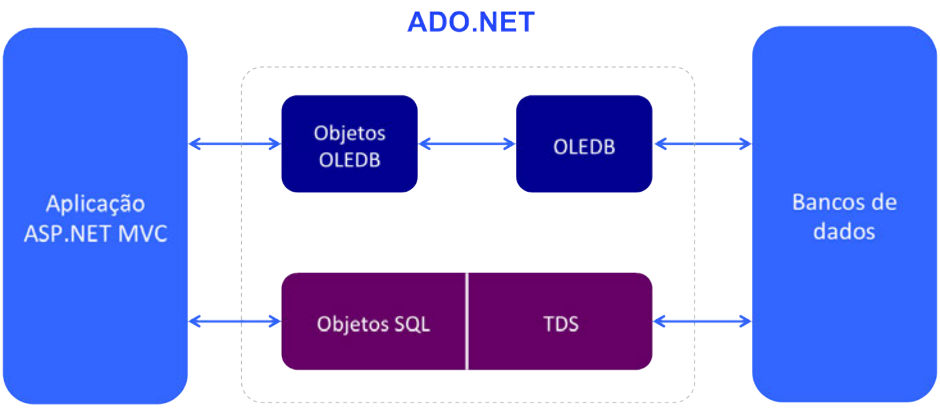


Figura 4.1: Visão conceitual do ADO.NET

Em aplicações ASP.NET MVC, você pode optar por criar seus repositórios de acesso a dados utilizando, de forma direta, as classes e recursos do ADO.NET. Neste caso, basta que você crie seu método de conexão ao banco e os demais métodos de acesso e execução das sentenças SQL.

Se você está sendo apresentado a este *set* de recursos através deste texto, os trechos de código apresentados na sequência poderão ajudar a entender o *modos operandi* deste modelo.

Se houvesse a necessidade de nos conectarmos a um banco de dados, levando-se em consideração o fato de que a *string* de conexão pudesse variar, por exemplo, poderíamos ter um método de conexão fictício "**ConectaNoBancoDeDados**". Ele utiliza a classe "**SqlConnection**" presente no *namespace* "**System.Data.SqlClient**", conforme apresenta a listagem 1.

Listagem 4.1 - Método fictício para conexão com o banco de dados via ADO.NET:

```
using System.Data.SqlClient;
public bool ConectaNoBancoDeDados(string stringDeConexao)
{
    SqlConnection objetoDeConexao = new SqlConnection(stringDeConexao);

    try
    {
        objetoDeConexao.Open();

        if(objetoDeConexao.State == ConnectionState.Open)
        {
            return true;
        }
    }
    catch(Exception ex)
    {
        throw new Exception(ex.Message)
    }
}
```

Desta forma, bastaria ir compondo sua(s) classe(s) de acesso a dados. Em um projeto ASP.NET MVC, estas estruturações deveriam (de acordo com a convenção do *framework*) ocorrer no diretório *Models*.

Vale lembrar neste momento que nosso objetivo aqui não é apresentar detalhes acerca do ADO.NET, mas sim, apresentar a abordagem como uma possibilidade dentro de aplicações ASP.NET MVC. Você pode encontrar mais informações acerca do *set* de recursos disponibilizados pelo ADO.NET através do link: *http://bit.ly/mvc-adonet*.

ORM's entrando em ação

Se você já possui alguma experiência com desenvolvimento de *software*, por certo já ouviu falar sobre um aspecto de suma importância para projetos de *software*: os ORM's (*Object Relational Model* em inglês e "Objeto Modelo-Relacional" em português).

ORM's são recursos (ferramentas) criados para facilitar aos desenvolvedores o processo de conexão por parte de suas aplicações, às fontes de dados. Em linhas gerais, o que estas ferramentas fazem é criar uma camada de abstração em relação ao

banco de dados, disponibilizando portanto um modelo em nível mais alto (programático) para acessar e manipular os dados.

ORM's apresentam características funcionais importantes que contribuem em grande escala para sua adoção em projetos de *softwares* modernos. Dentre estas características destacam-se:

- **Simplicidade**: talvez a principal característica proporcionada pelos ORM's é a simplificação do modelo de acesso aos dados. Graças a esta simplificação, desenvolvedores podem trabalhar com as aplicações sem necessariamente possuírem conhecimentos avançados de SQL e bancos de dados;

- **Produtividade**: como o desenvolvedor acessa e manipula os dados através de um modelo programático simplificado, desenvolvedores tendem a ser mais produtivos, uma vez que o modelo de desenvolvimento lhe é familiar;

- **Redução de código**: como o acesso aos dados é feito em um nível mais alto, automaticamente o número de linhas de código escritas para acessar os dados reduz drasticamente. Você poderá constatar este fato na medida em que avançamos com a criação da aplicação **Cadê meu médico**;

- **Facilita a manutenção**: o modelo disponibilizado pelos ORM's facilita e muito a manutenção da aplicação, pois a manipulação das operações se dá através de código já conhecido (objetos e seus respectivos métodos);

- **Código elegante**: o fato de o modelo mesclado de sentenças SQL e chamadas de métodos de objetos dar lugar a um código mais legível e semântico torna o código fonte da aplicação mais elegante.

Para justificar as afirmações apresentadas na lista anterior, utilizaremos exemplos práticos. Imagine, por exemplo, que precisamos adicionar um novo cliente na tabela fictícia de `Clientes`. Se optarmos pela utilização das classes nativas da plataforma .NET (ADO), poderíamos ter um trecho de código semelhante ao apresentado pela listagem 2.

Listagem 4.2 - Conectando ao banco de dados e adicionado um novo cliente via ADO:

```
string stringDeConexao =
        "string para conexão com o banco de dados aqui";
```

```
SqlConnection objetoDeConexao = new SqlConnection(stringDeConexao);
SqlCommand objetoDeComando = objetoDeConexao.CreateCommand();
string sql = "INSERT INTO Clientes (NomeCompleto, Email)
                values (@NomeCompleto, @Email)";
objetoDeComando.CommandText = sql;
objetoDeComando.CommandType = CommandType.Text;
objetoDeComando.Parameters.Add("@NomeCompleto", NomeCompleto.Text);
objetoDeComando.Parameters.Add("@Email", Email.Text);
objetoDeComando.ExecuteNonQuery();
```

O código apresentado pela listagem 2, como é possível observar, é bastante simples, entretanto, extenso e verboso. Para realizar a mesma operação utilizando um ORM para .NET, poderíamos ter um trecho de código semelhante ao apresentado pela listagem 3.

Listagem 4.3 - Conectando ao banco de dados e adicionando um cliente através de um ORM:

```
Clientes.Add();
Clientes.NomeCompleto = NomeCompleto.Text;
Clientes.Email = Email.Text;
Clientes.Save();
```

Tanto a listagem 2 quanto a listagem 3 apresentam códigos que executam a mesma operação: adicionar um novo cliente ao banco de dados. Não é difícil observar a diferença exorbitante entre as abordagens. Enquanto temos muito código na primeira, temos uma redução considerável na segunda. Enquanto temos muitos objetos em ação na primeira abordagem, temos bem poucos na segunda.

É importante observar, entretanto, que mesmo possuindo as vantagens já mencionadas neste capítulo, existem alguns gargalos relacionados aos ORM's e que, portanto, devem ser cuidadosamente analisados para não impactar de forma crítica durante o ciclo de vida da aplicação. O principal aspecto a ser considerado olhando-se por este prisma é a performance. Mas não se preocupe, voltaremos a discutir sobre isso quando estivermos trabalhando com o *model* de nossa aplicação exemplo.

Existe um grande número de ORM's disponíveis no mercado para o trabalho dentro da plataforma .NET. Você deverá analisar cuidadosamente cada opção para escolher aquele que melhor atende às necessidades do projeto. Evidentemente, alguns deles se destacam por serem reconhecidamente robustos, com boa documenta-

ção disponível e, também, por possuírem suporte. Em função disso, são amplamente utilizados em projetos mundo afora.

Para a aplicação **Cadê meu médico**, utilizaremos o *Entity Framework* (EF), um ORM distribuído gratuitamente pela Microsoft e que tem sido amplamente utilizado em projetos .NET. Na ocasião em que este livro foi escrito, a versão estável mais atual do EF era a 4.0.

4.3 O *Entity Framework*

EF, como o próprio nome sugere, é um *framework* do tipo ORM que permite tratar e manipular dados como classes e objetos de domínio. Por ser desenvolvido, mantido e disponibilizado pela Microsoft, ele se integra de forma otimizada às tecnologias disponíveis na plataforma .NET com performance, segurança e robustez. Por ser esta integração nativa, desenvolvedores podem utilizar também de forma natural dois grandes recursos da .NET *framework*: LINQ e expressões Lambda para recuperar e manipular os dados necessários à aplicação.

Você pode saber mais sobre o Entity Framework através do link:

http://bit.ly/mvc-ef

Código primeiro?

Uma das vantagens proporcionadas por boa parte dos ORMs e propositalmente não mencionada nas seção anterior é a possibilidade de escolher a forma de trabalho do ORM. Isto é, vamos nos basear primeiro no banco de dados físico ou o banco de dados que será gerado será um reflexo de nossa estrutura de classes? Nesta seção discutiremos um pouco sobre as diferenças, vantagens e desvantagens entre estas abordagens.

O famoso *code first*

Imagine poder criar uma estrutura bem definida de classes (Clientes, Usuários, Fornecedores etc.) cujos atributos e métodos (Codigo, Nome, Razão Social etc.) poderão atuar em um momento posterior, como estruturas primárias (tabelas, chaves primárias, estrangeiras etc.) na criação da estrutura física de banco de dados no servidor. Legal, não?

Esta é uma das características funcionais do *Entity Framework* e é conhecido como *code first* (código primeiro, em português). Para que esta ideia torne-se mais

clara, vamos à prática. Criaremos um projeto paralelo, em nada relacionado ao **Cadê meu médico**. Faremos desta forma porque para **Cadê meu médico** utilizaremos outra estratégia.

Com Visual Studio já em execução, crie um novo projeto ASP.NET MVC 4 (seguindo os passos já apresentados nos capítulos anteriores a este). Sugestivamente nomearemos esta aplicação de exemplo como "**AplicacaoComCodeFirst**". Vale observar que você precisará estar com Microsoft SQL Server 2012 Express instalado para completar este exercício. O projeto que estamos utilizando para exemplificar o funcionamento do *code first* é do tipo *internet application*. Ele utiliza *view engine* Razor e não utiliza projeto de testes. A *solution explorer* deste projeto pode ser visualizada pela figura 4.2.

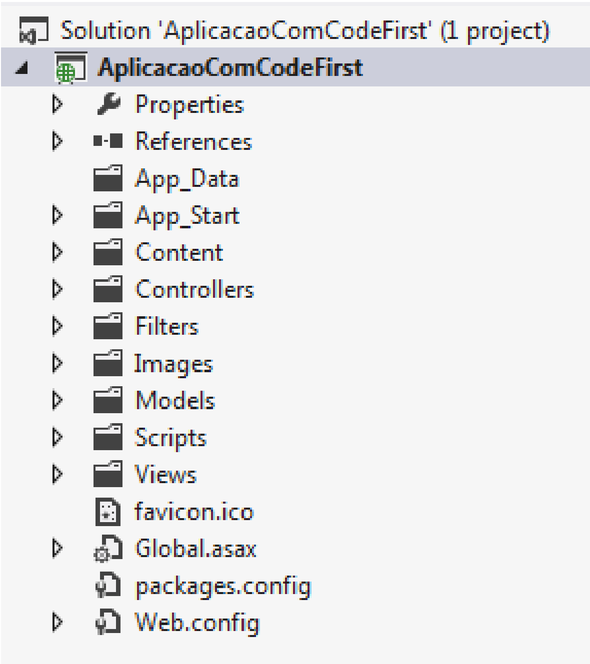


Figura 4.2: Solution explorer do projeto AplicacaoComCodeFirst

Por se tratar de um projeto do tipo *internet application*, quando executada, a aplicação tratá um *template* padrão (semelhante àquele apresentado no capítulo 1).

Você conhece o SQL Server 2012?

O Microsoft SQL Server 2012 é a versão mais atual da plataforma de dados relacionais da Microsoft. Através dela, é possível gerenciar qualquer ocorrência relacionada aos bancos de dados relacionais, incluindo ferramentas para mineração e extração de dados não convencionais, business intelligence [1], interface gráfica para administração (SQL Management Studio), entre diversos outros recursos.

A versão *Express* do SQL Server 2012 é disponibilizada gratuitamente e suporta bancos de dados com capacidade de até 10 GB. Para saber mais, você pode seguir o link:

http://bit.ly/mvc-sqlexpress

Um bom exemplo para que possamos entender de forma clara e prática o funcionamento do mecanismo de *code first* com *Entity Framework* dentro da plataforma .NET em projetos ASP.NET MVC é uma fração das funcionalidades de *blog*, ou seja, *posts* e suas respectivas categorias. Desta forma, imaginando a estrutura mais simples possível para este tipo de funcionalidade, poderemos ter a seguinte relação: uma categoria pode possuir múltiplos *posts* enquanto um *post* pode estar atrelado a apenas uma categoria.

Além da cardinalidade 1 para *n* configurada pela relação apresentada no parágrafo anterior, temos notoriamente visíveis duas entidades: `Categorias` e `Posts`. Muito embora esta seja uma relação simples, ela nos possibilitará apresentar alguns importantes recursos do *framework* MVC, além é claro, de apresentar o *code first* em ação.

Criando as classes POCO

Quando falamos em *code first*, automaticamente falamos em classes POCO. Isso porque não é possível dissociar uma coisa da outra.

POCO é o acrônimo para *Plain Old CLR Object*. O termo é uma derivação do termo POJO (*Plain Old Java Object*), originalmente utilizado pela comunidade Java, como você pode imaginar. A ideia deste tipo especial de classe é ser um agente independente de *frameworks* e/ou componentes dentro da plataforma .NET. Assim, classes podem herdar seus comportamentos, interfaces podem ser implementadas e ainda, atributos podem ser persistidos.

Tendo em mente a importância das classes POCO principalmente para o contexto de ORM's (claro, especialmente para o EF), iniciaremos nossas atividades criando nossas duas classes POCO principais: `Categorias` e `Posts`. No momento posterior, elas servirão de base para que o EF implemente as entidades no banco de dados da aplicação.

Desta forma, a primeira ação neste sentido será criar a classe `Categorias`. Para isso, basta adicionar uma nova classe à pasta *Model* da aplicação e nomeá-la com a palavra `Categorias`, conforme ilustra a figura 4.3.

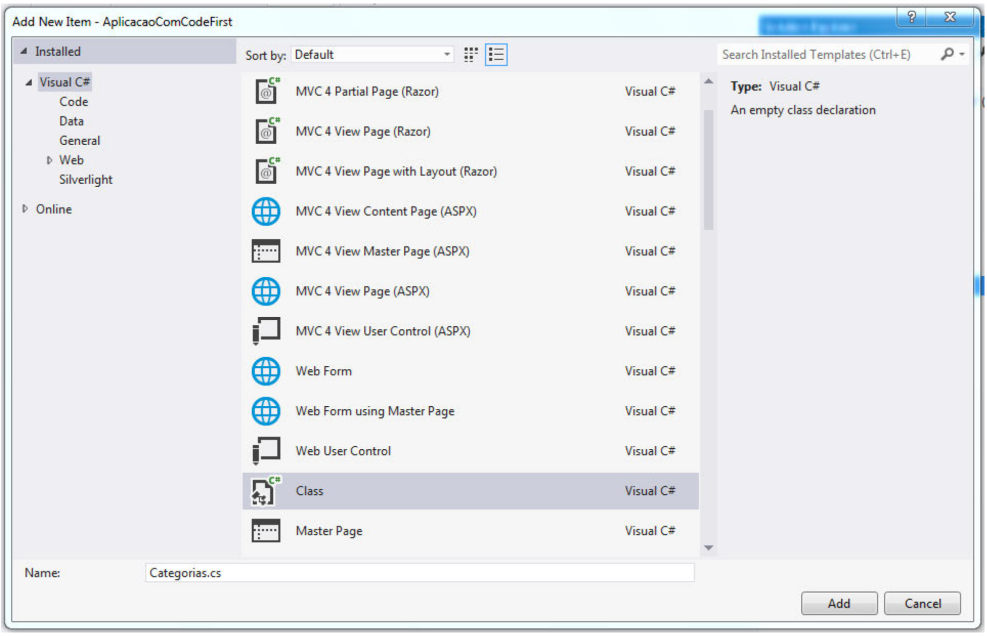


Figura 4.3: Adicionando uma classe ao diretório *Model*

Uma classe com estrutura básica será automaticamente gerada pelo Visual Studio. Substitua este conteúdo por aquele apresentado pela listagem 4.

Listagem 4.4 - Classe POCO para 'Categorias':

```
public class Categorias
{
    [Key]
    public int CategoriaID { get; set; }
```

```
    public string Categoria { get; set; }

    public string Descricao { get; set; }

    public virtual ICollection<Posts> Posts { get; set; }
}
```

Como você pode perceber, a estrutura da classe é bem simples e dispensaria comentários para um desenvolvedor com certa experiência no trato com ORM's. Entretanto, é possível que as observações a seguir sejam úteis para você que não se enquadra na situação apresentada há pouco.

- `Atributo [Key]`: indica ao Entity Framework que a propriedade imediatamente na sequência (no caso, `CategoriaID`) deverá ser considerada chave primária no momento da geração do banco de dados no servidor de destino. É possível parametrizar este atributo conforme a necessidade para emplacar comportamentos específicos (como a adição ou não da cláusula `identity`). Para a criação do nosso exemplo, manteremos o padrão utilizado pelo EF para esta geração;
- `public virtual ICollection<Posts> Posts { get; set; }`: em uma ferramenta tradicional para composição de banco de dados relacionais, conseguimos explicitar os relacionamentos de 1 para muitos ou muitos para muitos de forma bem simples. Para expressar esse tipo de relacionamento através de *code first*, utilizamos uma coleção de objetos tipada; justamente o que estamos fazendo com a linha em destaque neste tópico. Estamos dizendo ao EF que uma `Categoria` deverá suportar múltiplos `Posts`. Legal, não?!

Para que a classe `Categorias` possa existir, é preciso que `Posts` também exista. Desta forma, você pode conferir através da listagem 5 a estrutura da classe `Posts`.

Listagem 4.5 - Classe POCO que representa 'Posts':

```
public class Posts
{
    [Key]
```

```
    public long PostID { get; set; }

    public string TituloPost { get; set; }

    public string ResumoPost { get; set; }

    public string ConteudoPost { get; set; }

    public DateTime DataPostagem { get; set; }

    public int CategoriaID { get; set; }

    [ForeignKey("CategoriaID")]
    public virtual Categorias Categorias { get; set;
}
```

A observação válida em relação à listagem 5 fica por conta da última propriedade. Diferentemente da classe `Categorias` onde possuímos uma coleção de objetos (sim, uma `Categoria` pode possuir vários `Posts`), aqui, possuímos uma instância de `Categoria`, já que, pela definição da regra de negócio, um `Post` pode pertencer a apenas uma `Categoria`. O atributo `[ForeignKey("CategoriaID")]`, como você já deve estar imaginando, informa ao EF que a propriedade imediatamente a seguir assumirá o comportamento de chave estrangeira e que, portanto, será o elo com outra tabela — em nosso caso, a tabela `Posts`.

Para que a *engine* do Entity Framework possa ser capaz de gerar a estrutura de banco de dados com base nas classes POCO, é preciso que exista uma classe de amarração, na qual são informados parâmetros cruciais no processo de geração da estrutura final. Esta classe foi convencionada pelo EF como **classe de contexto**. Nela, implementamos algumas ações que são fundamentais para que tudo funcione de forma adequada. Confira o código apresentado pela listagem 6.

Listagem 4.6 - Classe de contexto para o Entity Framework:

```
public class BlogContext : DbContext
{
    public DbSet<Posts> Posts { get; set; }

    public DbSet<Categorias> Categorias { get; set; }
}
```

A classe de contexto a qual nos referimos foi criada nos mesmos moldes das demais apresentadas até aqui. Em se tratando de modelos de dados, evidentemente está dentro do diretório **Models**.

Esta é a estrutura mais simplificada possível para a classe de contexto. Aqui estamos "dizendo" ao EF, através das diretivas **DbSet**, que as classes `Posts` e `Categorias` deverão ser criadas no banco de dados de destino.

Uma pergunta que naturalmente surge neste ponto é: "O banco de dados será criado em qual servidor?".

Esta é uma pergunta interessante e a resposta está diretamente atrelada ao modelo de convenções do qual falamos no segundo capítulo. Por padrão, seguindo uma convenção do EF e não do ASP.NET MVC, se nenhuma string de conexão com o servidor de destino for informada para o ORM, no momento da criação, o EF criará a estrutura de banco de dados no SQL LocalDB ou Express Edition. Aquele que estiver disponível — se ambos estiverem disponíveis, LocalDB será a escolha.

Se for preciso apontar a criação da estrutura de banco de dados para um servidor específico, uma das possibilidades disponíveis para se realizar tal operação é através da implementação de um método construtor na classe `BlogContext`. A listagem 7 apresenta esta alteração.

Listagem 4.7 - Classe de contexto adaptada para utilizar um servidor específico:

```
public class BlogContext : DbContext
    {
        public BlogContext() : base("name=BlogContext")
        {
            Database.Connection.ConnectionString =
                @"data source=FABRCIOSANC36FC\SQLEXPRESS;
            initial catalog=BlogBDLivro; Integrated Security=SSPI";
        }

        public DbSet<Categorias> Categorias { get; set; }

        public DbSet<Posts> Posts { get; set; }
    }
```

A mudança em relação à listagem 6 anterior é simples: adicionamos o método construtor da classe de contexto e, em seu interior, apontamos a *engine* do Entity

Framework para o servidor local `FABRCIOSANC36FC\SQLEXPRESS` para a criação do banco de dados `BlogBD`.

Testando a funcionalidade do code first

Agora que já possuímos a infraestrutura necessária para a aplicação exemplo, podemos dar o passo no sentido de ver o Entity Framework *code first* funcionando. Para isso, criaremos dois novos *controllers*: `Categorias` e `Posts`. Para cada um deles, criaremos operações de CRUD (acrônimo para *Create*, *Read*, *Update* e *Delete*) utilizando um recurso muito interessante disponível desde a versão 2 da *framework* MVC: *Scaffold*.

> **Mais sobre Scaffold**
>
> *Scaffold* é o recurso do ASP.NET MVC que permite criar operações de criação, leitura, atualização e remoção de registros no banco de dados de forma automatizada, utilizando boas práticas de desenvolvimento e a *view-engine* padrão (a saber, Razor). Com base na estrutura de banco de dados mapeada, o ASP.NET MVC é capaz de gerar as *actions* e respectivas *views* para possibilitar a realização das operações de CRUD.

No caso específico no *code first*, no momento em que apontarmos para o *framework* o contexto de dados para o qual o CRUD deverá ser gerado, automaticamente o banco de dados físico será gerado no servidor de destino.

É importante observar neste ponto que, para que as classes POCO possam ser incorporadas ao *core* do projeto, será preciso executar um *build* para ela. Este é o primeiro passo.

Na sequência, adicione o *controller* `Categorias` à aplicação (você já sabe como fazer, basta clicar com o botão direito sobre o diretório "*Controller*", adicionar um novo item e nomeá-lo na sequência). Após executar estes passos, o Visual Studio lhe apresentará uma nova janela. Para o caso do *controller* `Categorias`, no box "*Scaffold options*", para a opção "*Template*", selecione na lista suspensa a opção "*MVC controller with read/write actions and views, using Entity Framework*". Na opção seguinte (*Model class*), selecione a opção "Categorias", e na seguinte (*Data context class*), a opção "BlogContext". A figura 4.4 apresenta a seleção destas opções.

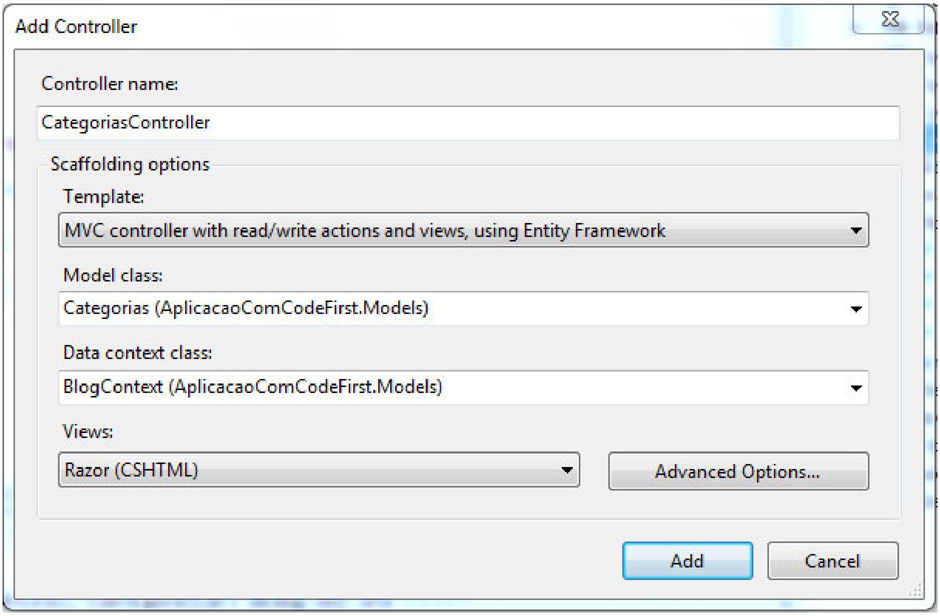


Figura 4.4: Adicionando o controller 'Categorias' com Scaffold

Para o *controller* `Posts`, a mesma sequência de procedimentos deverá ser realizada.

Ao concluir estes passos, você poderá visualizar o Visual Studio criar códigos de forma automática, tanto para o *controller* quanto para as *views* associadas. Além disso, se configurarmos a guia "*Database Explorer*" do Visual Studio para nos conectarmos à instância de SQL Server apontada no código da listagem 7 após a execução da aplicação, podemos comprovar a criação, por parte do Entity Framework, do banco de dados `BlogBDLivro` (a figura 4.5 apresenta este fato).

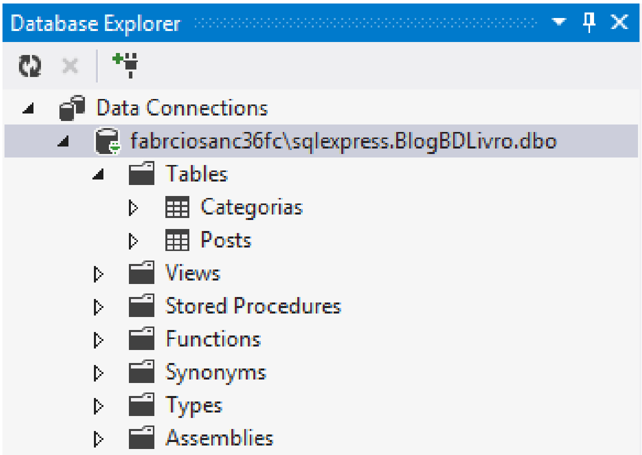


Figura 4.5: Banco de dados criado via *code first*

Para testar o funcionamento do *scaffold*, basta executar a aplicação e chamar as URLs respectivas para `Categorias` ("/Categorias") e `Posts` ("/Posts"), conforme ilustra a figura 4.6.

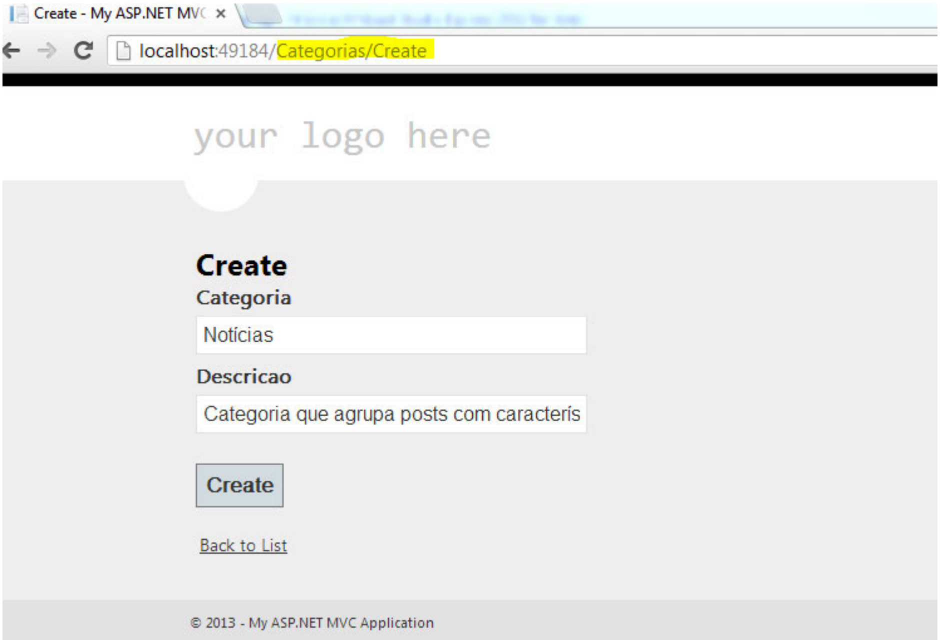


Figura 4.6: Cadastrando uma nova categoria com scaffold

Desafiando: que tal criar agora um modelo de classes do tipo "muitos para muitos" via *code first* e, utilizando o recurso de *scaffold*, gerar operações de CRUD baseada nelas?

4.4 O modelo primeiro?

Se na abordagem anterior partíamos do código fonte para o banco de dados físico, quando temos um ambiente onde o modelo vem primeiro (como é o caso do *model first*), somos obrigados a pensar que tudo parte de um modelo predefinido para chegar, em um momento posterior, ao banco de dados.

Com Entity Framework também possuímos esta possibilidade. Neste caso, basta que criemos um modelo vazio (nos moldes do que faremos na seção seguinte) e que, conforme necessário, adicionemos as entidades. Assim, em um momento posterior, o EF com base no modelo criado gerará os elementos no banco de dados físico.

Como este formato de trabalho está longe de ser o mais utilizado, não entraremos em maiores detalhes neste livro. Entretanto, existem algumas boas fontes de consulta internet afora. Algumas delas encontram-se listadas a seguir:

- **Model First with Entity Framework — MSDN**: http://bit.ly/mvc-ef-modelfirst
- **Entity Framework Tutorial**: http://bit.ly/mvc-ef-tutorial

4.5 Banco de dados primeiro?

A seção anterior, apresentou um dos modelos possíveis para o trabalho com o Entity Framework — o *code first*. Como você pôde comprovar naquele formato, saímos de um modelo programático e chegamos há um modelo de banco de dados físico.

Nesta seção, apresentaremos outra forma de trabalho com o EF — o *database first* que, de forma oposta ao modelo visto anteriormente, nos permite sair de um modelo físico de banco de dados para chegarmos a um modelo programático.

Voltando à aplicação "Cadê meu médico?"

Como você deve se lembrar, no capítulo 3, elaboramos e apresentamos a estrutura da aplicação **Cadê meu médico**. Nesta estruturação, um dos aspectos apresentados foi o seu DER, onde elucidamos de forma gráfica e conceitual a estrutura futura do banco de dados da aplicação (ver figura 3.2). Vale notar que, em relação ao banco de dados, nada foi implementado ainda.

Como estamos falando neste momento de *database first*, parece fazer total sentido implementar, antes de qualquer outra coisa, o banco de dados da aplicação **Cadê meu médico**. A listagem 8 apresenta o *script* SQL que deu origem ao banco de dados ao qual nos referimos.

Para facilitar a utilização, estamos disponibilizando o *script* online, através do link:

http://bit.ly/mvc-script-bd

Listagem 4.8 - Script SQL que deu origem ao banco de dados da aplicação 'Cadê meu médico':

```
CREATE DATABASE CadeMeuMedicoBD
GO

USE CadeMeuMedicoBD
GO
```

```
CREATE TABLE Usuarios
(
    IDUsuario BIGINT IDENTITY(1,1) NOT NULL,
    Nome VARCHAR(80) NOT NULL,
    Login VARCHAR(30) NOT NULL,
    Senha VARCHAR(100) NOT NULL,
    Email VARCHAR(100) NOT NULL,

    PRIMARY KEY(IDUsuario)
);
GO

CREATE TABLE Medicos
(
    IDMedico BIGINT IDENTITY(1,1) NOT NULL,
    CRM VARCHAR(30) NOT NULL,
    Nome VARCHAR(80) NOT NULL,
    Endereco VARCHAR(100) NOT NULL,
    Bairro VARCHAR(60) NOT NULL,
    Email VARCHAR(100) NULL,
    AtendePorConvenio BIT NOT NULL,
    TemClinica BIT NOT NULL,
    WebsiteBlog VARCHAR(80) NULL,
    IDCidade INT NOT NULL,
    IDEspecialidade INT NOT NULL,

    PRIMARY KEY(IDMedico)
);
GO

CREATE TABLE Especialidades
(
    IDEspecialidade INT IDENTITY(1,1) NOT NULL,
    Nome VARCHAR(80) NOT NULL,

    PRIMARY KEY(IDEspecialidade)
);
GO

CREATE TABLE Cidades
(
```

```
    IDCidade INT IDENTITY(1,1) NOT NULL,
    Nome VARCHAR(100) NOT NULL,

    PRIMARY KEY(IDCidade)
);
GO

ALTER TABLE Medicos
ADD CONSTRAINT fk_medicos_cidades
FOREIGN KEY(IDCidade)
REFERENCES Cidades(IDCidade)
GO

ALTER TABLE Medicos
ADD CONSTRAINT fk_medicos_especialidades
FOREIGN KEY(IDEspecialidade)
REFERENCES Especialidades(IDEspecialidade)
GO

INSERT INTO Cidades (Nome) VALUES ('Blumenau')
INSERT INTO Cidades (Nome) VALUES ('São José do Rio Preto')
GO

INSERT INTO Especialidades (Nome) VALUES ('Cardiologista')
INSERT INTO Especialidades (Nome) VALUES ('Neurologista')
GO

INSERT INTO Usuarios (Nome, Login, Senha, Email)
VALUES ('Administrador',
        'admin',
        '40BD001563085FC35165329EA1FF5C5ECBDBBEEF',
        'admin@cdmm.com')
GO
```

A figura 4.7 apresenta o resultado obtido ao executarmos o código apresentado pela listagem 8 no SQL Server 2012 Express.

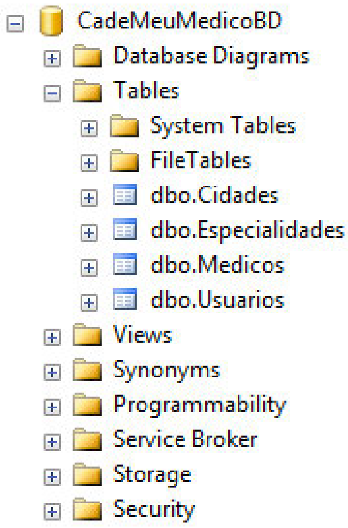


Figura 4.7: Server explorer do SQL Server Management Studio

Agora que já possuímos o banco de dados físico para a aplicação, estamos aptos a dar o próximo passo, isto é, criar seu mapeamento via Entity Framework para que a aplicação possa utilizá-lo. Para isso, adicionarei à pasta *Models* um novo tipo de arquivo, chamado "*ADO Entity Data Model*", conforme ilustra a figura 4.8.

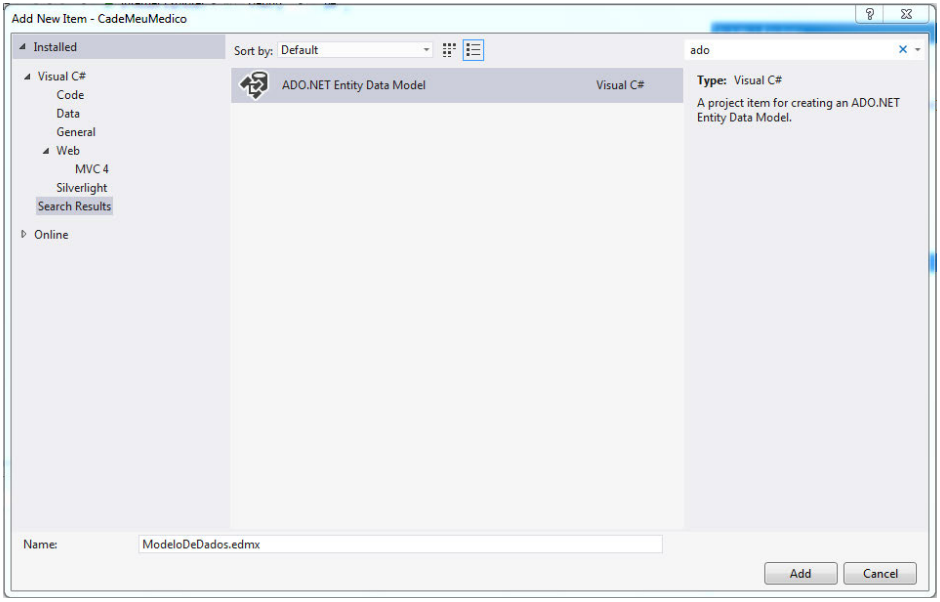


Figura 4.8: Selecionando a opção adequada para a criação do modelo de dados

Ao realizar tal procedimento, você estará automaticamente iniciando o processo de mapeamento requerido pelo EF. Como parte deste procedimento, uma nova janela solicitando o tipo de mapeamento que será realizado (figura 4.9) será apresentada. Para a construção deste exemplo, você deverá selecionar a primeira opção, isto é, "*Generate from database*" e prosseguir.

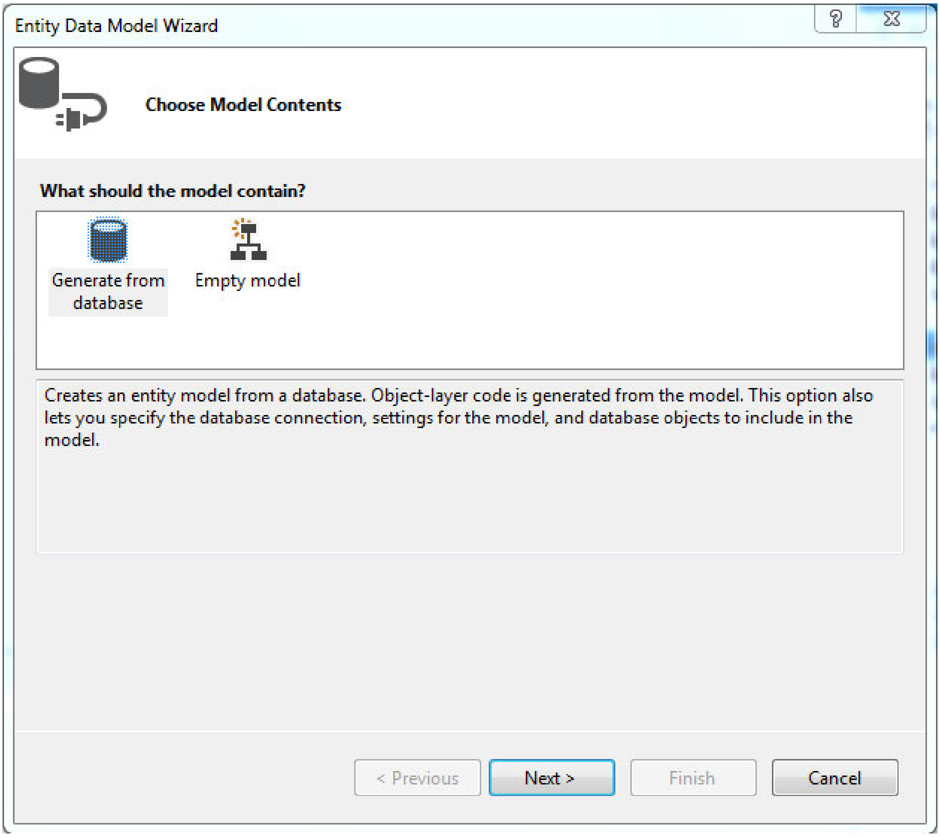


Figura 4.9: Selecionando o tipo de mapeamento que será realizado

A próxima janela solicita informações de *string* de conexão. O que faremos aqui é apontar o EF para o banco de dados que criamos recentemente. Para isso, clique no botão "*New connection*". Nela, você deverá informar alguns dados de conexão, como endereço para o servidor de banco de dados, usuário e senha (se possuir) para acesso e o nome do banco de dados ao qual deseja se conectar. A figura 4.10 apresenta a tela devidamente preenchida.

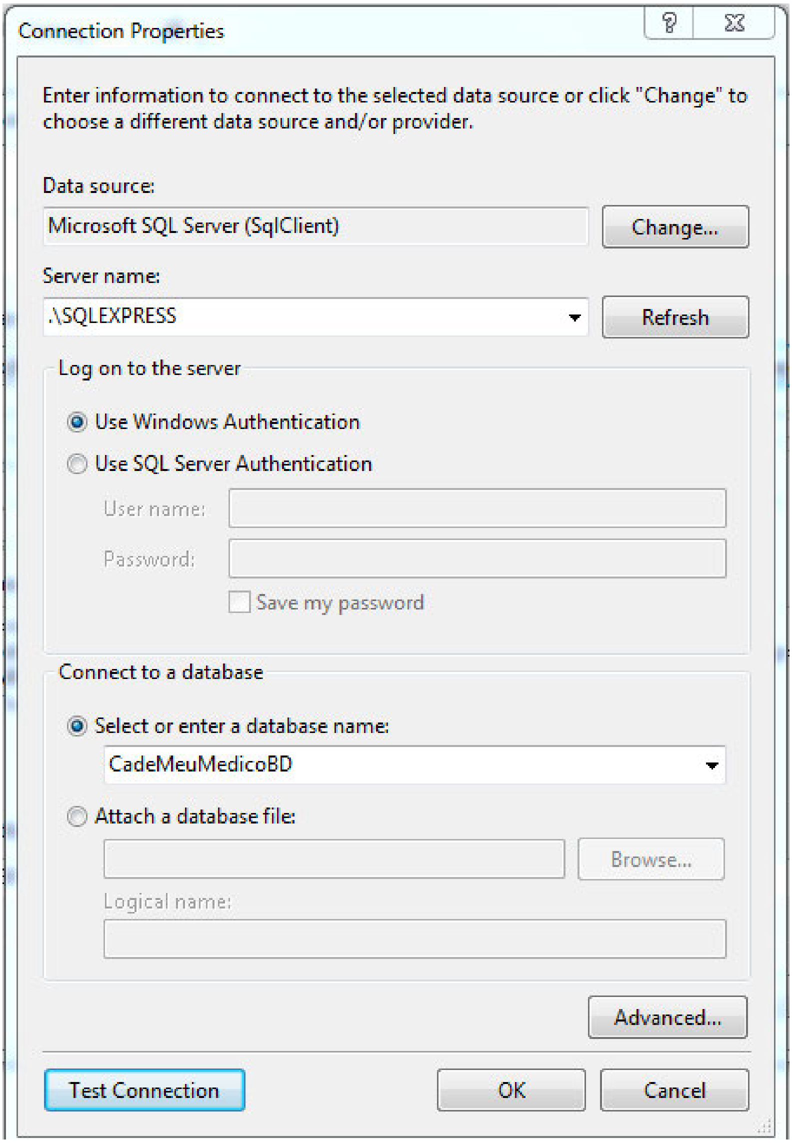


Figura 4.10: Adicionando informações de conexão

Ao clicar em *OK* você será levado de volta à tela anterior, mas agora, com a *string* de conexão já selecionada. Basta nomear a variável que será salva (e posteriormente referenciada) no arquivo `Web.config` da aplicação e prosseguir (ver figura 4.11).

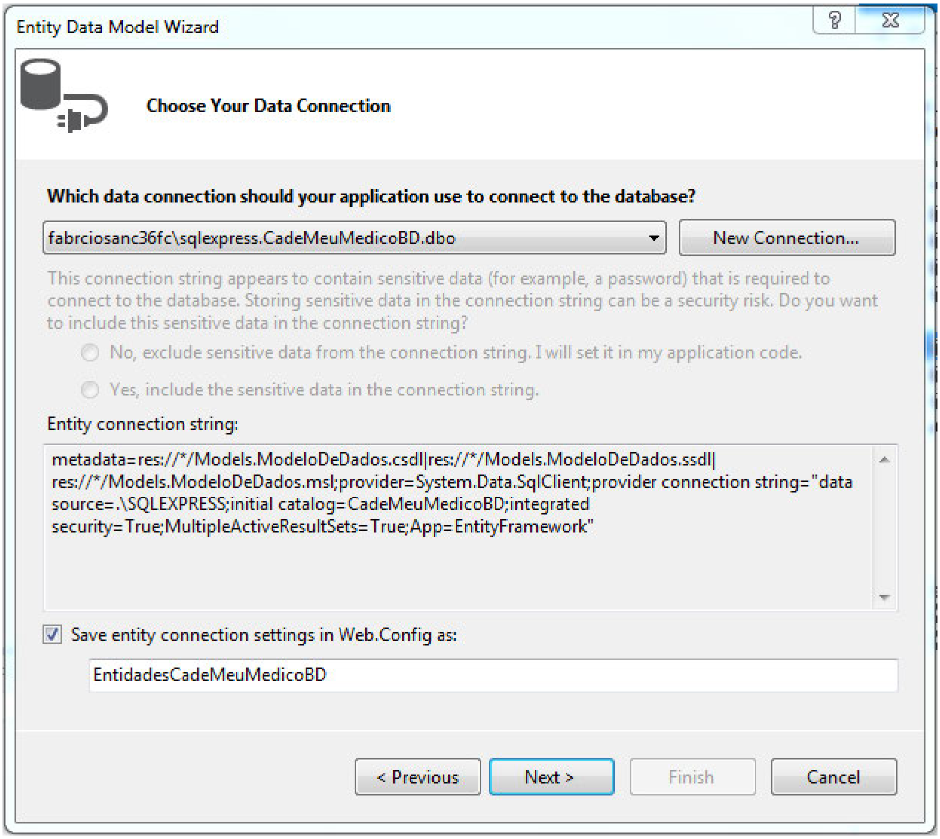


Figura 4.11: Selecionando string de conexão e nomeando a variável para Web.config

A próxima tela solicitará que você selecione os objetos do banco de dados que serão mapeados. Em nosso caso, selecionaremos todas as tabelas, conforme sugere a figura 4.12. Você também deverá nomear o *namespace* ao qual estão agrupados os arquivos do modelo. Utilizaremos o nome `CadeMeuMedicoBDModel`. Após isso, clique em "*Finish*".

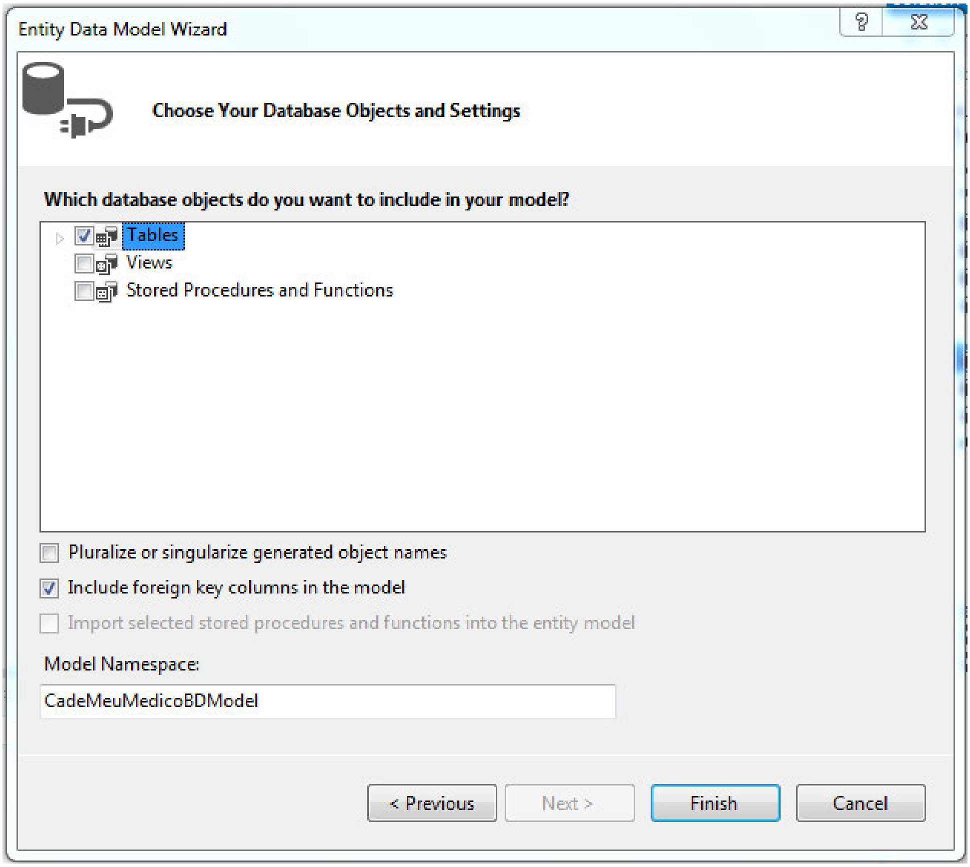


Figura 4.12: Selecionando os objetos a serem mapeados do banco de dados

Ao finalizar o processo, o Entity Framework irá criar toda a estrutura de classes necessária para que se possa utilizar o banco de dados em um nível mais alto. Se tudo correu bem, você deverá estar visualizando uma imagem semelhante à apresentada pela figura 4.13.

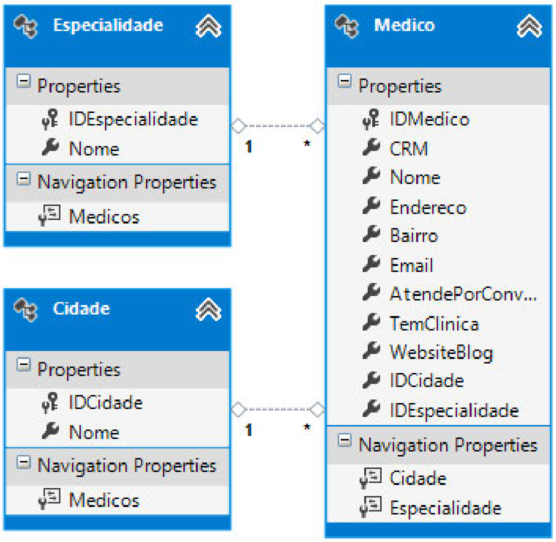


Figura 4.13: Modelo de classes gerado pelo Entity Framework

Além disso, algumas modificações estruturais no projeto poderão ser claramente notadas através da *solution explorer*. No diretório "*References*", por exemplo, você perceberá a adição de uma nova DLL, a saber, `System.Data.Entity`. A mudança mais importante entretanto, está evidentemente na pasta *Models*. Isso porque o EF adiciona automaticamente um arquivo com a extensão **.edmx** e este grande arquivo é o objeto final do mapeamento realizado. Agrupam-se, portanto, todas as classes necessárias para o trabalho com o banco de dados.

A figura 4.14 apresenta a estrutura do arquivo edmx gerado. Com um duplo clique sobre as classes, é possível visualizar suas estruturas, que seguem fielmente o modelo físico do banco de dados.

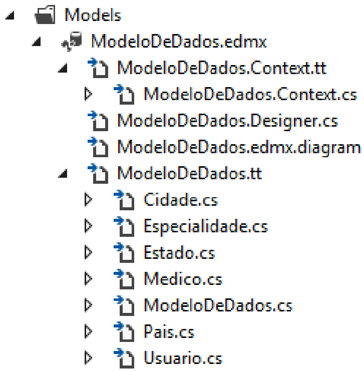


Figura 4.14: A estrutura do arquivo EDMX

Outra mudança que pode ser percebida na estrutura da aplicação, mas que não é visível através da *solution explorer*, é a adição da *string* de conexão no arquivo `Web.config`, conforme ilustra a listagem 9.

Listagem 4.9 - Adição da string de conexão no arquivo Web.config pelo Entity Framework:

```
<connectionStrings>
    <add name="EntidadesCadeMeuMedicoBD"
        connectionString="metadata=res://*/Models.ModeloDeDados.csdl|
        res://*/Models.ModeloDeDados.ssdl|
        res://*/Models.ModeloDeDados.msl;provider=System.Data.SqlClient;
        provider connection string=&quot;
        data source=.\SQLEXPRESS;initial catalog=CadeMeuMedicoBD;
        integrated security=True;
        MultipleActiveResultSets=True;App=EntityFramework&quot;"
        providerName="System.Data.EntityClient" />
</connectionStrings>
```

Pronto. Neste momento, temos tudo o que precisamos para conseguir trabalhar com o banco de dados tendo como foco o modelo recém-gerado. Para este capítulo, já reunimos toda informação necessária para seguir com os próximos passos. À medida que a aplicação for ganhando forma, você entenderá como se dá o processo de interação com o modelo que criamos aqui.

4.6 Model first x Code first: Quando utilizar um ou outro?

Nas seções anteriores, apresentamos os dois modelos de trabalho mais usuais quando o ORM em questão é o Entity Framework. Levando-se em consideração o fato de que ambos os modelos são funcionais e objetivam a mesma coisa (isto é, manipular dados), uma pergunta natural surge na cabeça de grande parte dos desenvolvedores: "quando utilizar a abordagem X ou Y?"

Como é de se esperar para perguntas deste tipo, a resposta mais prudente é: **depende**.

Existem diversos aspectos que devem ser considerados em um projeto de software ao se optar por determinada tecnologia e/ou metodologia. Alguns dos que devem ser considerados são: robustez, conectividade, segurança, performance e produtividade. Isso sem falar na preferência do gestor do projeto atrelado ao *expertise* de seu time.

Quando falamos de estratégia de acesso a dados utilizando EF, os aspectos mencionados também são decisivos. Assim, se não podemos dar uma resposta definitiva para a pergunta inicial desta seção, podemos ao menos apresentar vantagens e desvantagens de cada modelo para que você possa reunir insumos para escolher de forma mais assertiva a estratégia de acesso a dados para sua aplicação.

Code first: vantagens e desvantagens

Quando falamos em um modelo em que o código da aplicação poderá gerar a estrutura do banco de dados, imediatamente algumas características positivas já nos vêm a cabeça, certo? Vamos a elas:

- **Isolamento do código fonte da aplicação em relação ao banco de dados**: esta é, sem dúvida, a grande vantagem do EF *code first*. Como o modelo é todo baseado em classes e seus respectivos objetos, é possível que desenvolvedores consigam criar aplicações com estruturas de bancos de dados complexas sem escrever uma linha sequer de SQL (*Structured Query Language*);

- **Produtividade**: como o modelo de desenvolvimento proporcionado pelo *code first* é bem familiar, de forma geral, é possível obter boa produtividade por parte dos desenvolvedores;

- **Código limpo**: a criação das classes POCO ajudam a manter o código limpo,

uma vez que os desenvolvedores podem seguir os padrões de desenvolvimento de seus projetos;

- **Controle completo de código**: como as classes POCO são criadas e mantidas pelo próprio desenvolvedor, o controle sobre aquilo que é gerado no banco de dados é bem mais fácil de ser realizado. Classes geradas automaticamente tendem a ser de difícil manutenção;
- **Simplicidade**: de forma geral, o trabalho com *code first* tende a ser mais simples. Isso porque não existe a necessidade de se manter um arquivo **.edmx**. Qualquer problema gerado neste arquivo implicará necessariamente em problemas com a *engine* de acesso a dados.

Mas nem tudo são flores. *Code first* possui um "problema" intrínseco, proveniente do seu esquema de trabalho que limita sua utilização em boa parte dos projetos: de forma geral, ele pode ser aplicado apenas a novos projetos, já que, dependendo da complexidade do banco de dados de aplicações já existentes, fica inviável gerar classes POCO manualmente para obter a equivalência no banco de dados físico.

Dica: Se um novo projeto for seu objeto de estudo, considere utilizar o modelo *code first*. Os resultados deste tipo de abordagem para esta natureza de projeto costumam render excelentes frutos.

Database first: vantagens e desvantagens

É claro que existem muitos novos projetos de software. A todo instante nos deparamos com eles. Entretanto, existem projetos que precisam passar por complexos processos de migração, por exemplo, e, para este tipo de cenário, *database first* se enquadra perfeitamente. Assim, ao optar por modelo de trabalho, os desenvolvedores poderão usufruir dos seguintes benefícios:

- **Isolamento completo das atividades**: se o sistema já possui um banco de dados criado e/ou mantido por DBA's, utilizar o modelo *database first* permitirá separar ainda mais as responsabilidades do projeto, uma vez que a atualização do modelo baseado nas alterações realizadas diretamente no banco de dados funcionam muito bem;
- **Reduz drasticamente o tempo de mapeamento de bancos pré-existentes**: *database first* reduz de forma drástica o tempo dispensado ao mapeamento de bancos de dados preexistentes;

- **Personalização de classes POCO**: muito embora as classes POCO sejam geradas de forma automática pelo Entity Framework, é possível personalizá-las através de classes parciais ou até mesmo através do template de classes disponibilizado pelo EF;
- **Possibilidade de múltiplos modelos no mesmo projeto**: sim, é possível possuir diferentes mapeamentos, de diferentes bancos de dados, na mesma aplicação.

O principal problema relacionado ao modelo disponibilizado por *database first* reside na concentração de tudo dentro de um único arquivo (o **edmx**). Conforme o banco de dados de origem cresce em tamanho e complexidade, o arquivo também o faz, o que dificulta a manutenção. Em função disso, na grande maioria das vezes, é preciso que as máquinas que manipulam estes arquivos possuam boas configurações de memória.

Dica: De forma geral, o modelo *database first* tende a ser melhor aplicável em situações em que existem aplicações com seus bancos de dados já em funcionamento (migração, reengenharia etc.), mas, diferentemente do modelo *code first* que pode ser aplicado na maioria esmagadora das vezes apenas em novas aplicações, se você deseja possuir controle total das ocorrências no banco de dados, *database first* pode ser aplicável também para novos projetos. Assim, no final das contas, este modelo acaba se tornando mais flexível.

4.7 Adicionando atributos de validação nos modelos

A framework .NET disponibiliza através do namespace *System.ComponentModel.DataAnnotations* vários atributos de validação que podem ser em modelos de classes do Entity Framework e Linq To SQL.

Adicionando esses atributos de validação nos *models* (modelos) da nossa aplicação, o MVC fará essa validação de forma transparente, sem necessidade de codificação adicional por parte do programador para a validação básicas sobre o *model*.

Anteriormente, utilizamos o modelo *database first* para gerar os *models* da nossa aplicação, ou seja, os *models* são automaticamente gerados, baseado na estrutura do banco de dados . Uma boa prática é não colocar os atributos de validação diretamente nas classes geradas automaticamente pelo Visual Studio. Isso porque sempre que o modelo do Entity Framework for atualizado com base no banco de dados,

as classes serão recriadas, e com isso qualquer modificação feita anteriormente será perdida.

Seguindo essa boa prática, não iremos adicionar diretamente na classe gerada pelo Visual Studio os atributos de validação. Criaremos uma classe de metadados contendo as validações e, utilizando o recurso *Partial Class* ou *Classes Parciais*, vamos vincular a classe de metadado com a classe criada automaticamente.

Classes Parciais

Dentre vários recursos disponibilizados pela plataforma .NET, encontramos as Partial Class ou Classes Parciais. Graças às classes parciais podemos dividir a definição de uma classe em vários arquivos físicos. A keyword `partial` deverá ser utilizada na definição da classe para "dizer" que a classe é parcial.

Todas as classes geradas automaticamente utilizando Entity Framework recebem a keyword *partial* na sua definição. Devido a esse recurso conseguimos estender essas classes sem alterar o arquivo físico que é passível de modificação pelo Entity Framework.

Para criar as validações referentes ao *model* Médico, clique com o botão direito do mouse na pasta *Models* da *Solution Explorer* e clique em *Add*. Na janela que será apresentada selecione o tipo *Class*, e defina o nome da classe igual a `MedicoMetadado`. A figura 4.15 representa esse processo.

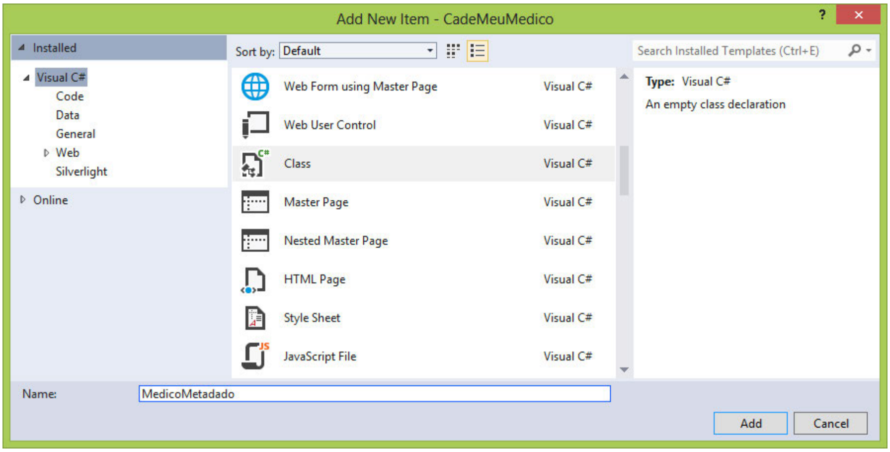


Figura 4.15: Adicionando a classe de metadado para o model Medico

Após a criação, o arquivo será aberto automaticamente pelo Visual Studio. A listagem 10 representa o código que deverá ser adicionado na classe MedicoMetadado.cs.

Repare que o arquivo `MedicoMetadado.cs` possui a definição das classes `MedicoMetadado` e a classe parcial `Medico`.

Na classe parcial `Medico` utilizamos o atributo *MetadataType* para informar qual o tipo da classe que possui os metadados que serão utilizados, e é com esses atributos que informamos que a classe *MedicoMetadado* deverá ser utilizada.

A classe `MedicoMetadado` possui as mesmas propriedades da classe `Medico`, afinal é nela que colocaremos as validações das propriedades. Em cada propriedade, utilizamos os atributos necessários para fazer a validação que desejamos.

Em todos os exemplos estamos utilizando só dois atributos, o atributo `Required` define que aquela propriedade é obrigatória, e já definimos nele a mensagem de erro que será apresentada caso a propriedade não esteja informada. Além da validação de campo obrigatório, podemos utilizar o atributo `StringLenght` para validar o tamanho dos textos informados, e caso o tamanho ultrapasse o limite informado, definimos também a mensagem de erro que será apresentada.

Listagem 4.10 - Definição dos Metadados do model Medico:

```
namespace CadeMeuMedico.Models
{
```

```
[MetadataType(typeof(MedicoMetadado))]
public partial class Medico
{

}

public class MedicoMetadado
{
    [Required(ErrorMessage="Obrigatório informar o CRM")]
    [StringLength(30, ErrorMessage="O CRM deve possuir no
    máximo 30 caracteres")]
    public string CRM { get; set; }

    [Required(ErrorMessage = "Obrigatório informar o Nome")]
    [StringLength(80, ErrorMessage = "O Nome deve possuir no máximo
     80 caracteres")]
    public string Nome { get; set; }

    [Required(ErrorMessage = "Obrigatório informar o Endereço")]
    [StringLength(100, ErrorMessage = "O Endereço deve possuir no
    máximo 100 caracteres")]
    public string Endereco { get; set; }

    [Required(ErrorMessage = "Obrigatório informar o Bairro")]
    [StringLength(60, ErrorMessage = "O Bairro deve possuir
        no máximo 60 caracteres")]
    public string Bairro { get; set; }

    [Required(ErrorMessage = "Obrigatório informar o E-mail")]
    [StringLength(100, ErrorMessage = "O E-mail deve possuir no
     máximo 100 caracteres")]
    public string Email { get; set; }

    [Required(ErrorMessage = "Obrigatório informar se Atende
        por Convênio")]
    public bool AtendePorConvenio { get; set; }

    [Required(ErrorMessage = "Obrigatório informar se Tem Clínica")]
    public bool TemClinica { get; set; }

    [StringLength(80, ErrorMessage = "O Website deve possuir no
```

```
        máximo 80 caracteres")]
        public string WebsiteBlog { get; set; }

        [Required(ErrorMessage = "Obrigatório informar a Cidade")]
        public int IDCidade { get; set; }

        [Required(ErrorMessage = "Obrigatório informar a
            Especialidade")]
        public int IDEspecialidade { get; set; }
    }
}
```

Vimos até aqui as possíveis formas de criação dos modelos da aplicação, além de adicionar atributos de validação as propriedades do modelo. Mas nossa aplicação não possui só o *model* Medico — ainda temos Usuários, Cidades e Especialidades. Repita o processo apresentado e faça as classes de metadados referente a cada modelo da aplicação.

Lembre-se que, caso seja necessário, os códigos fontes completos da aplicação estão no github:

http://bit.ly/mvc-livrocodigofonte

Mas é um excelente exercício você mesmo criar todas as entidades.

Capítulo 5

Controllers: Adicionando comportamento a nossa aplicação

O universo de desenvolvimento de software disponibiliza diferentes abordagens para a construção das aplicações (sobre o que já discutimos um pouco no primeiro capítulo deste livro). Tais abordagens podem variar em função de diversos aspectos, sendo os dois principais: o paradigma funcional da linguagem (orientada a objetos, funcional, procedural etc.) e as metodologias e ferramentas associadas para o trabalho conjunto com as linguagens e IDE's.

Neste contexto, podemos concluir que, evidentemente, existem diferentes formas para que se possa implementar os comportamentos desejados para as aplicações. Como você deve se lembrar, no modelo de programação proposto pelo ASP.NET Web Forms, por exemplo, tais comportamentos estavam diretamente ligados (via *code-behind*) a componentes de servidor que eram predefinidos pela Microsoft. Tal modelo é conhecido por muitos como "programação orientada a eventos" (herança de plataformas de desenvolvimento como Delphi, VB, dentre outras).

Quando falamos em ASP.NET MVC, referimo-nos necessariamente à implementação de comportamentos para as aplicações através de elementos do *framework* conhecidos como *controllers* (em português brasileiro, "Controladores"). Evidente que o nome *controller* não foi atribuído por acaso. Seu nome resume de forma sintética, porém perfeita, sua atuação dentro do padrão arquitetural MV**C**.

Neste capítulo discutiremos de forma mais aprofundada a utilização dos *controllers* no contexto de aplicações construídas através do ASP.NET MVC. Ao final, deverão estar claros para você os seguintes aspectos relacionados a estes elementos:

- O que são *controllers* e sua importância nos projetos ASP.NET MVC;
- O que são *actions* e sua relação com os *controllers*;
- Entender o comportamento ideal dos *controllers* em aplicações bem construídas;
- Modo de ativação e chamada de *controllers*;
- Criar *views* a partir de *actions* presentes nos *controllers*;
- *Actions* que podem não dar origem a *views* mas que são muito importantes.

Compreender o funcionamento dos *controllers* e suas *actions* é de fundamental importância para a construção de aplicações utilizando *ASP.NET MVC*. Definimos no capítulo anterior os *models* (modelos) da nossa aplicação, chegou o momento de adicionarmos um pouco de vida a ela.

Vamos começar?!

5.1 Vamos 'controlar' a aplicação?

Um orquestrador de operações diversas. Esta é uma boa definição para os *controllers* dentro da *framework* ASP.NET MVC, já que são eles que agrupam todos os comportamentos da aplicação.

Como você deve se lembrar, no capítulo 2 falamos um pouco sobre o modelo de rotas implementado pelo ASP.NET MVC. Naquela ocasião, mencionamos o fato de que, ao receber determinada requisição, o MVC procuraria o *controller* que responderia para ela, determinaria qual *action* a processaria e então, geraria a visualização para o cliente.

Em termos técnicos, um *controller* nada mais é do que uma classe que herda comportamentos de outra (a saber, **Controller**) disponível na BCL. Cada método disponível nesta classe especial é chamado de *Action*. As *Actions* são, efetivamente, a menor unidade de ação dentro de um *controller*. São elas que encapsulam os comportamentos que são invocados a todo momento, pela classe controladora.

5.2 Entendendo o papel dos Controllers e Actions

Fazendo uma analogia simples, podemos dizer que os *controllers* serão os mediadores das conversas entre as visões da aplicação (*views*) e os modelos (*models*).

Os *controllers* serão responsáveis por receber e transformar as requisições enviadas pelas *views* em informações que serão utilizadas pelas regras de negócio da aplicação, ou seja, os *models*. O contrário também é verdadeiro, os *controllers* têm a responsabilidade de selecionar a *view* correta para apresentação de informações ao usuário.

Ao receber uma requisição, os controllers devem efetuar alguma ação para que a requisição seja processada, essas ações são as *actions*. As actions são métodos definidos no controller, e o controller pode ter várias actions.

Para exemplificar a utilização dos *controllers* e *actions*, criaremos em nossa solução de exemplo (**Cadê meu médico**), um *controller* fictício com algumas *actions*. Mas antes de iniciarmos, é altamente recomendável que você entenda em sua plenitude o conceito de roteamento (já apresentado no capítulo 2).

Lá você pôde entender que o roteamento padrão do MVC é definido por `{controller}/{action}/{id}`. Em nosso exemplo, queremos que as seguintes URLs estejam respondendo na aplicação:

- http://localhost/mensagens/bomdia
- http://localhost/mensagens/boatarde

O processo de criação de novos controladores é simples, basta que você repita os passos já informados anteriormente, isto é, clique com o botão direito do mouse no diretório *Controllers* (na *Solution Explorer*) e na sequência, no menu que será apresentado, escolha as opções `Add > Controller`.

Uma nova janela onde podemos definir alguns parâmetros e propriedades do novo *controller* se mostrará. Você deverá fornecer como entrada para o campo *Controller Name* o valor `MensagensController` e, para o grupo *Scaffold options* no

campo *Template*, a opção *Empty MVC controller*. Ao final. a janela com os valores configurados deve estar parecida com a apresentada pela figura 5.1.

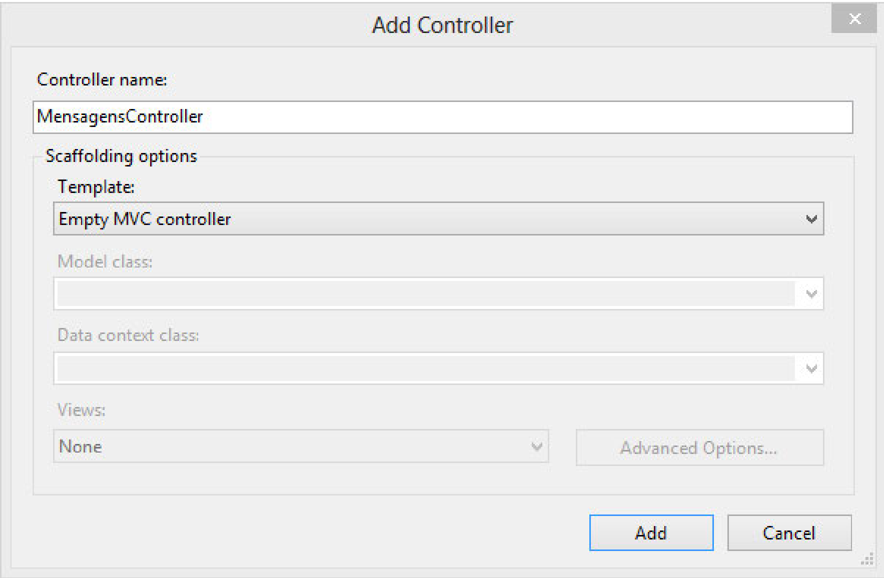


Figura 5.1: Criando o Controller Mensagens

Após isso, você notará que o Visual Studio abrirá o arquivo recém-criado e que no interior de nosso *controller* `MensagensController` uma *action* chamada `index` já estará criada. Assim, seu *controller* deverá estar parecido com o apresentado pela listagem 1.

Listagem 5.1 - Código padrão do MensagensController:

```
public class MensagensController : Controller
{
    //
    // GET: /Mensagens/

    public ActionResult Index()
    {
        return View();
    }
```

```
}
```

O que faremos agora é excluir a *action* `index` predefinida e criar novas duas novas, chamadas `BomDia` e `BoaTarde` respectivamente. Os códigos que compõem tais *actions* deverão apresentar alguma informação para o usuário. Estas serão as funções de ambas.

Não se preocupe, o tipo de retorno `ActionResult` e a diretiva de retorno `Content` que utilizaremos em nossos exemplos serão discutidas em maiores detalhes à medida que avançamos neste capítulo.

A listagem 2 apresenta o código das *actions* que executarão as tarefas que o usuário final solicitará e que você deverá adicionar ao seu *controller* para a visualização do exemplo.

Listagem 5.2 - Actions que retornam mensagens para o usuário:

```
public class MensagensController : Controller
{
    public ActionResult BomDia()
    {
        return Content("Bom dia... hoje você acordou cedo!");
    }

    public ActionResult BoaTarde()
    {
        return Content("Boa tarde... não durma na mesa do trabalho!");
    }
}
```

Pronto. Nosso projeto já pode ser executado. Compile-o utilizando a tecla `F5`. Seu navegador padrão será trazido à execução e irá apresentar uma mensagem de erro dizendo que "*Não foi possível encontrar o recurso*". Muito embora a mensagem de erro seja emitida, ela já era esperada. Isso acontece porque ainda não criamos o *controller* padrão configurado no roteamento do nosso projeto.

Neste momento, para que você possa visualizar o exemplo em execução, desconsidere a mensagem de erro e, na barra de endereço de seu navegador, complete a URL apresentada com os valores `/mensagens/bomdia` ou `/mensagens/boatarde`. Perceba que estamos solicitando especificamente um `/{controller}/{action}` e, com isso, o MVC faz o roteamento para o *controller* e *action* que solicitamos e que podem ser aplicados.

Veja através da figura 5.2 um exemplo de uma mensagem sendo exibida ao usuário.

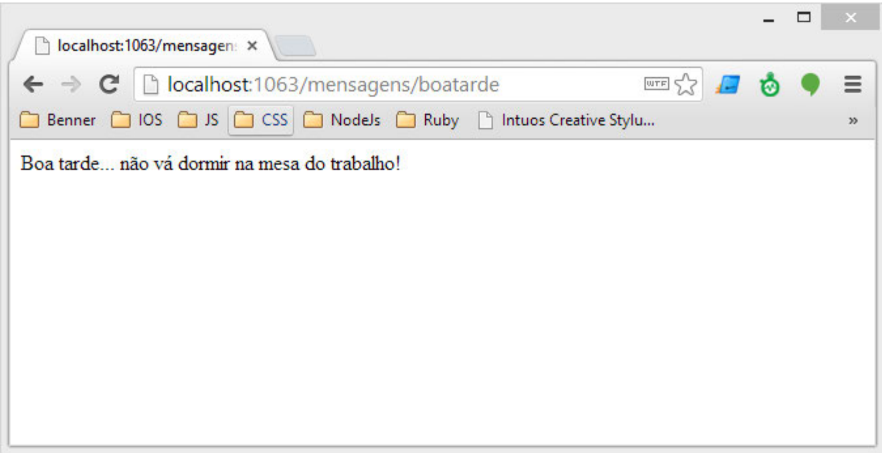


Figura 5.2: Usuário visualizando a mensagem definida na action

Como você pôde perceber, quando digitamos a URL http://localhost:1063/mensagens/boatarde , obtivemos a resposta "*Boa tarde... não vá dormir na mesa do trabalho*". Isso ocorreu porque, ao receber a requisição, o *controller* `mensagens` direcionou o fluxo de processamento para a *action* solicitada, isto é, `boatarde`. Por sua vez, a *action* `boatarde` executou a operação que lhe era devida e enviou um retorno de **conteúdo** (note a utilização da diretiva *Content*), neste caso, uma *string* simples.

ActionResult e ContentResult, quais as diferenças?

ActionResult é o tipo de retorno mais genérico existente dentro do *framework* ASP.NET MVC. Ele suporta qualquer tipo de resultado que possa ser retornado, sendo os mais comuns: **JsonResult**, **ContentResult**, **EmptyResult**, **FileResult**, **JavaScriptResult**, **RedirectResult**. Cada um dos mencionados nada mais é do que a extensão (através de herança) de ActionResult. Sua implementação é extremamente simples, contendo apenas um método para ser implementado na classe que a herda, haja vista que ActionResult é uma classe abstrata.

Desta forma, **ContentResult** trata-se de uma implementação extensiva da classe ActionResult, que permite formatar qualquer tipo de conteúdo que se deseja retornar, especificando inclusive, seu *ContentType*. Textos simples (como nosso exemplo) até sentenças mais complexas (como nodos de XML, por exemplo) podem ser retornados através de *Content*.

Agora que você já foi apresentado aos conceitos iniciais relacionados aos *controllers*, é chegado o momento de voltar as atenções para nossa aplicação de exemplo. Isso será feito na próxima seção.

5.3 Cadê meu médico: o que faremos?

A partir daqui, apresentaremos algumas formas práticas para o trabalho com *controllers*, utilizando exemplos reais (todos no contexto da aplicação **Cadê meu médico**). Apresentaremos os diferentes modelos de interação entre *views* e *controllers* e, ao final, você terá presenciado algumas das diversas formas de trabalho com *controllers*.

Como já mencionamos, existem diferentes formas para realizar a interação entre *controllers* e *views* em aplicações ASP.NET MVC. Neste capítulo, apresentaremos as principais (comumente utilizadas em projetos reais). Assim, para que você possa estruturar de forma ainda melhor seus estudos, listamos a seguir os principais aspectos relacionados aos *controllers* que veremos aqui. São eles:

- Como passar dados de um *controller* para uma *view*;
- Como passar dados de uma *view* para um *controller*;

- Passando dados de um formulário para *controller*;
- Comunicação de *controllers* com arquivos de repositórios;
- CRUDs utilizando o Scaffold do ASP.NET MVC.

Para que todas essas ações possam ser implementadas, criaremos CRUDs para cada um dos elementos da aplicação **Cadê meu médico** (você se lembra: Usuários, Médicos, Especialidades e Cidades).

5.4 Cadê meu médico: CRUDs

Iniciaremos o desenvolvimento do *backend* da aplicação *Cadê meu médico* criando o CRUD essencial para o funcionamento da aplicação, isto é, o cadastro de **Médicos**. Faremos desta forma porque é essencial compreendermos, neste momento, o fluxo de trabalho básico utilizando o ASP.NET MVC.

Através do cadastro de *Médicos* será possível entender o modelo através do qual enviamos e recebemos informações dos formulários HTML, além de criar toda a lógica básica nas *actions* do nosso *controller* para que o cadastro funcione de forma adequada. Muito embora tenhamos um capítulo inteiro dedicado ao estudo das *views*, já no cadastro de *Médicos* faremos algumas modificações nas *views* geradas automaticamente pela *framework* MVC.

Hora de começar a construção do cadastro de *Médicos*. Basta que você repita os passos já informados anteriormente, isto é, clique com o botão direito do mouse no diretório *Controllers* (na *Solution Explorer*) e na sequência, no menu que será apresentado, escolha as opções `Add > Controller`.

Na janela de criação do *controller*, nomeie-o como `MedicosController`. Não se esqueça da *CoC* associada aos *controllers* em relação aos seus nomes. Estes nomes devem possuir o sufixo *Controller*. Seguindo com a parametrização, na opção *Template*, selecione o valor *Empty MVC controller*. A janela completamente parametrizada deve se parecer com a figura 5.3.

Apesar do ASP.NET MVC possuir um mecanismo de *Scaffolding* poderoso que cria automaticamente *controllers*/*actions*/*views*, construiremos manualmente todo o *controller* e demais recursos necessários. Isso fará com que o entendimento de todo o fluxo de criação e funcionamento do MVC, seja facilitado.

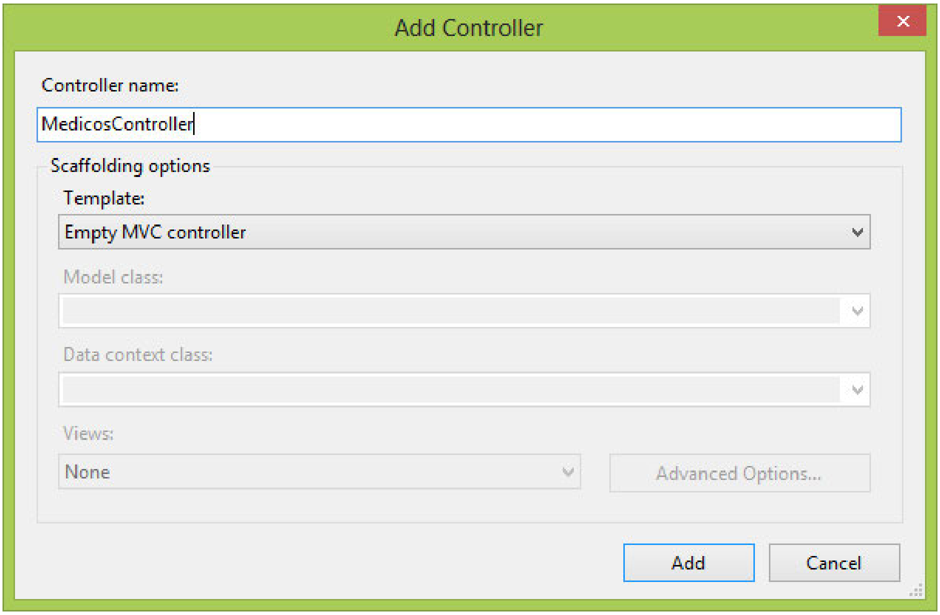


Figura 5.3: Criação do MedicosController

É importante notar o fato de que, muito embora estejamos utilizando o *template Empty MVC controller* para este primeiro *controller* de exemplo, ao ser criado, o `MedicosController` já traz em seu interior uma *action* padrão chamada *Index*. O código padrão do *controller* deve estar parecido com o da listagem 3.

Listagem 5.3 - Código padrão criado pelo Visual Studio:

```
public class MedicosController : Controller
{
    //
    // GET: /Medicos/

    public ActionResult Index()
    {
        return View();
    }
}
```

Quando o administrador do aplicativo navegar até a URL `/Medicos/`, será possível visualizar a lista de todos os médicos já cadastrados. Os modelos da nossa

aplicação foram criados utilizando *Entity Framework* e é através dele que obteremos a lista de médicos já cadastrados para que, em um momento posterior, possamos enviá-los para a *view*.

Evoluir é preciso, assim, continuando a construção do `MedicosController`, precisamos agora criar uma instância do objeto `EntidadesCadeMeuMedicoBD` que possibilitará todo o trabalho de comunicação dos nossos modelos (*models*) com o banco de dados da aplicação. Lembre-se de adicionar o namespace `CadeMeuMedico.Models` nos *usings* do *controller* `MedicosController`. Com a instância do contexto de entidades do *Entity Framework* criada, podemos na *action* `Index` obter uma lista com todos os Médicos cadastrados e, no instante seguinte, passar estes dados para a *view*. A listagem 4 representa isso.

Listagem 5.4 - Action Index retornando a lista de médicos cadastrados:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using CadeMeuMedico.Models;

namespace CadeMeuMedico.Controllers
{
    public class MedicosController : Controller
    {
        private EntidadesCadeMeuMedicoBD db =
                new EntidadesCadeMeuMedicoBD();

        public ActionResult Index()
        {
            var medicos = db.Medicos.Include(m => m.Cidade)
                        .Include(m => m.Especialidade).ToList();
            return View(medicos);
        }
    }
}
```

O código apresentado pela listagem 4 é simples. O aspecto a ser ressaltado aqui é o retorno da *action* `Index`. Como você pode perceber, estamos retornando uma lista (`.ToList()`) de todos os médicos (`.Medicos`) disponíveis no contexto (db).

Outro detalhe interessante na listagem 4 é a utilização dos métodos `.Include()`. A utilizar o método `.Include`, informamos ao *Entity Framework* que além do modelo (tabela) que estamos carregando, queremos obter seus relacionamentos, ou seja, nesse caso estamos listando os médicos e obtendo sua respectiva "Especialidade" e "Cidade".

Agora que já possuímos a *action* `Index` criada e funcionando, podemos adicionar a *view* que responderá visualmente à solicitação de `Index` no navegador — aquela que mostrará a lista de médicos cadastrados para o usuário final. Para realizar tal operação, faremos uma vez mais o uso do *scaffolding* da *framework* ASP.NET MVC.

Clique com o botão direito do mouse sobre o método *View* dentro da *action* Index, no menu de contexto que será exibido selecione a opção *Add View...*, a figura 5.4 demonstra esse processo.

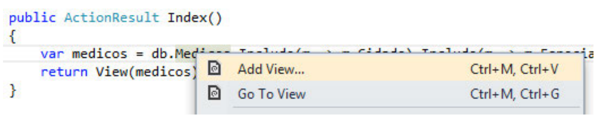


Figura 5.4: Adicionando uma nova View

Na janela que será apresentada, configuraremos os detalhes referentes a nova *view*. Configure a *view* da seguinte forma:

- *View name*: automaticamente o MVC vai sugerir o mesmo nome da *action*, nesse caso `Index`;
- *Create a strongly-typed view*: marque essa opção pois, através dela, criaremos uma *view* fortemente tipada, fato que facilitará nosso trabalho, permitindo a utilização direta do *model* na *view*;
- *Model class*: selecione o *model* que servirá de base para a criação da *view*. Nesse caso, sua opção deve ser por "Medico";
- *Scaffold template*: define qual template de *view* será criado. Neste caso, selecione a opção *List*. Como consequência, o MVC criará automaticamente uma *view* com uma tabela pronta para demonstrar os registros existentes;
- *Reference script libraries*: desmarque essa opção. Toda referência de script necessária para o funcionamento do aplicativo foi realizada na *Master Page*;

- *Use a layout master page*: essa opção deve ser selecionada para criar uma *view* que utilize a *Master Page* já definida anteriormente.

A figura 5.5 representa esse processo. Todas as *views* criadas no aplicativo a partir de agora seguirão este *template*, onde alteraremos somente as propriedades de *model class* e *Scaffold template*.

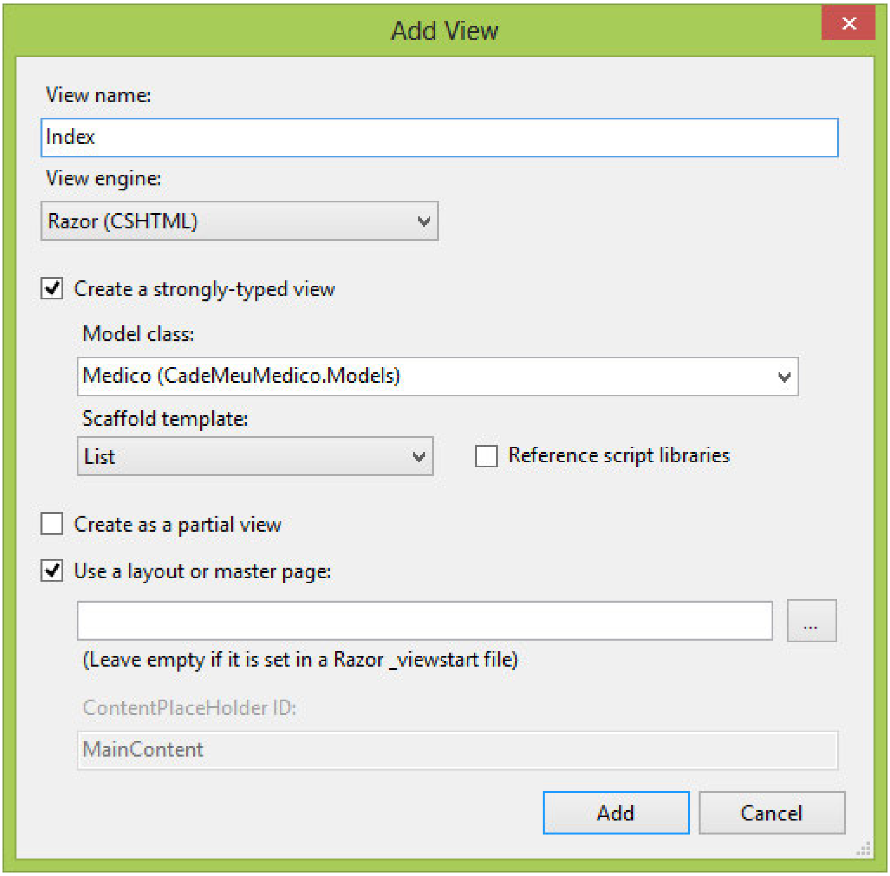


Figura 5.5: Adicionando uma nova View

Após clicar no botão *Add*, o Visual Studio criará e automaticamente abrirá a *view* para edição. A listagem 5 apresenta o código gerado para a *view* `Index`.

O próximo capítulo será dedicado integralmente à apresentação de detalhes relacionados às *views* e à sua engine padrão — o ASP.NET *Razor*. Não se preocupe

agora com as particularidades do *Razor* mencionadas, faremos poucas alterações só para ter um cadastro em pleno funcionamento.

Listagem 5.5 - Código padrão gerado pelo MVC pra a view:

```
@model IEnumerable<CadeMeuMedico.Models.Medico>

@{
    ViewBag.Title = %%Index%%;
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.IDMedico)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.CRM)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Nome)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Endereco)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Bairro)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Email)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.AtendePorConvenio)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.TemClinica)
```

```
        </th>
        <th>
            @Html.DisplayNameFor(model => model.WebsiteBlog)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.IDCidade)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.IDEspecialidade)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.IDMedico)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.CRM)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Nome)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Endereco)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Bairro)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Email)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.AtendePorConvenio)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.TemClinica)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.WebsiteBlog)
```

```
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.IDCidade)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.IDEspecialidade)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit",
                            new { /* id=item.PrimaryKey */ }) |
                @Html.ActionLink("Details", "Details",
                            new { /* id=item.PrimaryKey */ }) |
                @Html.ActionLink("Delete", "Delete",
                            new { /* id=item.PrimaryKey */ })
            </td>
        </tr>
}

</table>
```

Por padrão, o scaffold do MVC cria uma tabela com uma coluna para cada propriedade do modelo, nesse caso, temos um grid que irá mostrar todos os dados dos médicos. No caso da aplicação *Cadê meu Médico*, os dados que deveremos mostrar no grid devem ser apenas: Nome, Cidade e Especialidade.

Modifique o código da *view* Index removendo as colunas desnecessárias, mantendo somente as colunas para Nome, Cidade e Especialidade. A listagem 6 apresenta o código esperado. Repare também que, nas colunas Cidade e Especialidade, modificamos o valor apresentado. Originalmente o scaffold adicionou as propriedades IDEspecialidade e IDCidade, o que para o usuário não faz sentido — para ele a informação necessária é o nome referente a essas informações.

Na listagem 4 utilizamos o recurso `.Include()` do EntityFramework para carregar os relacionamentos do modelo, com isso fica fácil navegar entre os relacionamentos do modelo e apresentar outras informações ao usuário.

Listagem 5.6 - Listagem de médicos mostrando só os dados necessários:

```
@model IEnumerable<CadeMeuMedico.Models.Medico>

@{
    ViewBag.Title = %%Index%%;
```

```
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Nome)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.IDCidade)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.IDEspecialidade)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Nome)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Cidade.Nome)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Especialidade.Nome)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit",
                    new { /* id=item.PrimaryKey */ }) |
            @Html.ActionLink("Details", "Details",
                    new { /* id=item.PrimaryKey */ }) |
            @Html.ActionLink("Delete", "Delete",
                    new { /* id=item.PrimaryKey */ })
        </td>
    </tr>
```

```
}

</table>
```

Antes de compilar o projeto com o objetivo de testar o funcionamento da *view*, voltaremos à *master page* onde se encontra definida a estrutura de *layout* do aplicativo e adicionaremos um novo item ao menu principal para que este nos leve, então, até o cadastro de médicos.

Para isso, na *Solution Explorer*, abra o arquivo "Layout.cshtml" que se encontra no diretório *Views*. Note que na estrutura HTML da página, possuímos uma lista (<ul>) com a classe "nav" atribuída. É justamente nesta lista que os itens do menu devem ser adicionados. Como você poderá notar, ele já possui um link para a página inicial do site.

A listagem 7 mostra o código necessário para a criação do menu. Nele, você utilizará uma *HTML Helper* do *Razor* que cria um link no momento da renderização, ou seja, a tag <a> do HTML. Para que o *helper* chamado ActionLink possa funcionar corretamente, devemos informar 3 parâmetros: o texto a ser exibido pelo link, a *action* que será chamada por ele e o *controller* ao qual a *action* pertence.

Listagem 5.7 - Criação do item de menu para Médicos:

```
<ul class="nav navbar-nav">
    <li class="active"><a href="#">Home</a></li>
    <li id="menuMedicos">
        @Html.ActionLink("Médicos", %%Index%%, "Medicos")
    </li>
</ul>
```

Agora sim. Ao compilar o projeto, você verá que a aplicação exibe um item de menu chamado *Médicos*. Ao clicar no menu, você será redirecionado para a URL `/medicos`. Evidentemente, nenhum médico será exibido na página, uma vez que não adicionamos médicos ainda. O importante aqui é notar que já é possível ver o título da página, um link para criar novos médicos e o cabeçalho da tabela (grid). A figura 5.6 demonstra a página em execução.

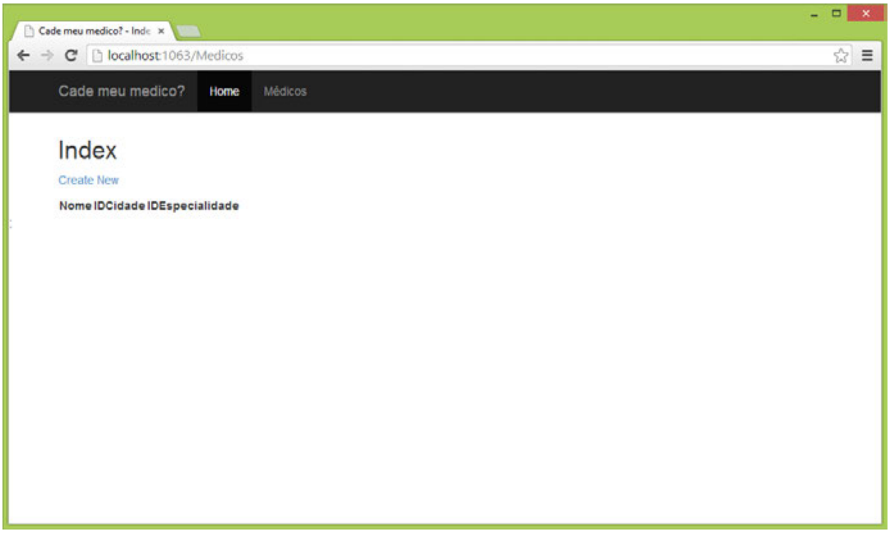


Figura 5.6: Cadastro de Médicos

Como já possuímos nossa primeira *action* e *view* construídas (onde o usuário poderá ver a lista de médicos cadastrados), estamos aptos a dar o passo seguinte, isto é, construir o formulário de cadastro de novos médicos. Esta tela de cadastro estará disponível através da URL `/Medicos/Adicionar`. Com base nesta URL e no *CoC* do MVC, sabemos que o nome de nossa próxima *action* deverá ser `Adicionar`.

Assim, no arquivo `MedicosController` na *Solution Explorer*, adicione uma nova *Action* chamada *Adicionar*. Esta *action* possui natureza um pouco diferente daquela que criamos nos passos anteriores: além de retornar uma *view* para o cadastro do médico, utilizaremos dois novos recurso que o *framework* MVC disponibiliza — o `ViewBag` e `SelectedList`.

Vimos na modelagem do sistema que os médicos possuem em suas informações básicas a Cidade de atuação e sua Especialidade. No formulário de cadastro do Médicos, essas informações serão apresentadas em um ComboBox (ou Dropdownlist). No caso de HTML, o ComboBox é representado pelo elemento `select`.

Podemos utilizar o `ViewBag` para transferir dados do *Controller* para a *View*. Ele é uma propriedade do tipo *dynamic*, por isso podemos criar propriedades dinamicamente. Em nossa *action* criamos duas propriedades, IDCidade e IDEspecialidade, cada uma com a lista de Cidades e Especialidades que mais adiante será apresentada ao usuário.

Nas propriedades dinâmicas da `ViewBag` retornarmos já o elemento que será apresentado na *View*. Para isso utilizamos o helper `SelectList` — você verá mais sobre os Helpers no próximo capítulo sobre *Views*.

A listagem 8 apresenta o código referente à *action* `Adicionar`.

Listagem 5.8 - Action adicionar:

```
public ActionResult Adicionar()
{
    ViewBag.IDCidade = new SelectList(db.Cidades, "IDCidade", "Nome");
    ViewBag.IDEspecialidade = new SelectList(db.Especialidades,
                                              "IDEspecialidade",
                                              "Nome");

    return View();
}
```

Da mesma forma que fizemos para adicionar a *view* `Index`, faremos também para criar a *view* `Adicionar`, com a diferença, entretanto, de que agora escolheremos o *template Create*.

Após efetuar a criação, o Visual Studio abrirá automaticamente o arquivo da *view*. Seu código deverá estar parecido com aquele apresentado pela listagem 9. Note que, no lugar de uma tabela para listar registros já cadastrados, o *template* `Create` gerou um formulário para o preenchimento dos dados do *model*.

Listagem 5.9 - View Adicionar:

```
@model CadeMeuMedico.Models.Medico

@{
    ViewBag.Title = "Adicionar";
}

<h2>Adicionar</h2>

@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>Medico</legend>
```

```
<div class="editor-label">
    @Html.LabelFor(model => model.IDMedico)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.IDMedico)
    @Html.ValidationMessageFor(model => model.IDMedico)
</div>

<div class="editor-label">
    @Html.LabelFor(model => model.CRM)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.CRM)
    @Html.ValidationMessageFor(model => model.CRM)
</div>

<div class="editor-label">
    @Html.LabelFor(model => model.Nome)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Nome)
    @Html.ValidationMessageFor(model => model.Nome)
</div>

<div class="editor-label">
    @Html.LabelFor(model => model.Endereco)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Endereco)
    @Html.ValidationMessageFor(model => model.Endereco)
</div>

<div class="editor-label">
    @Html.LabelFor(model => model.Bairro)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Bairro)
    @Html.ValidationMessageFor(model => model.Bairro)
</div>
```

```
<div class="editor-label">
    @Html.LabelFor(model => model.Email)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Email)
    @Html.ValidationMessageFor(model => model.Email)
</div>

<div class="editor-label">
    @Html.LabelFor(model => model.AtendePorConvenio)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.AtendePorConvenio)
    @Html.ValidationMessageFor(model =>
            model.AtendePorConvenio)
</div>

<div class="editor-label">
    @Html.LabelFor(model => model.TemClinica)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.TemClinica)
    @Html.ValidationMessageFor(model => model.TemClinica)
</div>

<div class="editor-label">
    @Html.LabelFor(model => model.WebsiteBlog)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.WebsiteBlog)
    @Html.ValidationMessageFor(model => model.WebsiteBlog)
</div>

<div class="editor-label">
    @Html.LabelFor(model => model.IDCidade)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.IDCidade)
    @Html.ValidationMessageFor(model => model.IDCidade)
</div>
```

```
        <div class="editor-label">
            @Html.LabelFor(model => model.IDEspecialidade)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.IDEspecialidade)
            @Html.ValidationMessageFor(model => model.IDEspecialidade)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", %%Index%%)
</div>
```

Analisando o código recém-criado, é possível notar que o formulário apresenta um campo chamado `IDMedico` para preenchimento. Veja, este campo refere-se a um *identity* no banco de dados, ou seja, este valor será gerado automaticamente pelo GBD (Gerenciador de Banco de Dados) quando chegar a requisição de gravação. Assim, faremos uma pequena alteração no HTML gerado para endereçar este "problema". Removeremos da *view* todo o código referente a esse campo no formulário. A listagem 10 apresenta o trecho de código que deverá ser removido.

Listagem 5.10 - Código que deve ser removido:

```
<div class="editor-label">
    @Html.LabelFor(model => model.IDMedico)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.IDMedico)
    @Html.ValidationMessageFor(model => model.IDMedico)
</div>
```

Faremos ainda outra pequena alteração no HTML gerado. Utilizaremos as propriedades da `ViewBag` para criar os Combos de nossa tela. Altere a linha `@Html.EditorFor(model => model.IDCidade)` para `@Html.DropDownList("IDCidade", String.Empty)`, e a li-

nha `@Html.EditorFor(model => model.IDEspecialidade)` para `@Html.DropDownList("IDEspecialidade", String.Empty)`.

Pronto. Agora que possuímos o fomulário pronto e funcionando, precisamos gravar os dados provenientes dele no banco de dados. Para isso, é preciso criar a *action* que receberá um *model* já com os dados que o usuário preencheu utilizando o formulário da *view*. Primeiro, é necessário compreender os verbos *GET* e *POST* do do protocolo HTTP. Apesar de mais verbos estarem disponíveis no HTTP (*DELETE*, *OPTIONS*, entre outros), para a nossa aplicação de exemplo utilizaremos apenas *GET* e *POST*.

Em linhas gerais, o verbo *GET* é utilizado para obter um recurso do servidor, enquanto o verbo *POST* serve para adicionar um novo recurso. Por recurso, entenda páginas, imagens, estilos, scripts, dados etc.

A listagem 11 demostra o código da *action* que receberá, em um parâmetro, o *model* preenchido pelo usuário na *view*. Além de validar o modelo e adicionar no banco de dados se nenhuma inconsistência for encontrada, note que a *action* também possui o nome `Adicionar`, entretanto, encontra-se decorada com o atributo *HttpPost*. Desta forma, pelo atributo decorativo da *Action*, o ASP.NET MVC consegue diferenciar os métodos com mesmo nome.

Listagem 5.11 - Action com atributo HttpPost:

```
public ActionResult Adicionar()
{
    ViewBag.IDCidade = new SelectList(db.Cidades, "IDCidade", "Nome");
    ViewBag.IDEspecialidade = new SelectList(db.Especialidades,
                                             "IDEspecialidade",
                                             "Nome");

    return View();
}

[HttpPost]
public ActionResult Adicionar(Medico medico)
{
    if (ModelState.IsValid)
    {
        db.Medicos.Add(medico);
        db.SaveChanges();
        return RedirectToAction(%%Index%%);
    }
```

```
        ViewBag.IDCidade = new SelectList(db.Cidades, "IDCidade",
                                                      "Nome",
                                                      medico.IDCidade);
        ViewBag.IDEspecialidade = new SelectList(db.Especialidades,
                                                 "IDEspecialidade",
                                                 "Nome",
                                                 medico.IDEspecialidade);
        return View(medico);
    }
```

O atributo *HttpPost* na *action* diz ao *framework* MVC qual *action* ele deve executar quando a requisição HTTP possuir o verbo *POST*. Isso é necessário porque a URL `/Medicos/Adicionar` é utilizada para mostrar o formulário e também para enviar os dados preenchidos. O que muda nas requisições são os verbos executados e os valores enviados para o servidor.

A listagem 11 também possui a *action* que será executada quando a requisição possuir o verbo *GET* e é ela quem retorna a página com o formulário. Apesar de o atributo *HttpGet* existir, ele não precisa ser adicionado porque *actions* sem atributo, por convenção, são consideradas *HttpGet*.

Estamos quase prontos para testar nosso primeiro cadastro. Antes porém, precisamos fazer uma pequena alteração no arquivo "Index.cshtml" que está na pasta `Views > Medicos` da *Solution Explorer*. O *framework* MVC cria todos os *templates* utilizando o idioma inglês. Assim, por inércia, ele sempre irá criar links apontando para *actions Create*, *Edit*, *Delete*. Como você deve se lembrar, substituímos `Create` por `Adicionar`. Assim, precisamos substituir na *view* as chamadas para `Create` por `Adicionar`.

No arquivo `Index.cshtml`, encontre o código `@Html.ActionLink("Create New", "Create")` e troque por `@Html.ActionLink("Adicionar novo médico", "Adicionar")`. O que estamos fazendo é trocar o texto do link: e em vez de ele apontar para a action `Create`, agora aponta para action `Adicionar`.

Compile seu projeto, navegue até a página de Países, e clique no botão *Adicionar novo médico*. A página `/Medicos/Adicionar` será carregada; informe só o nome do médico e clique no botão `Create`. A figura 5.7 representa esse processo.

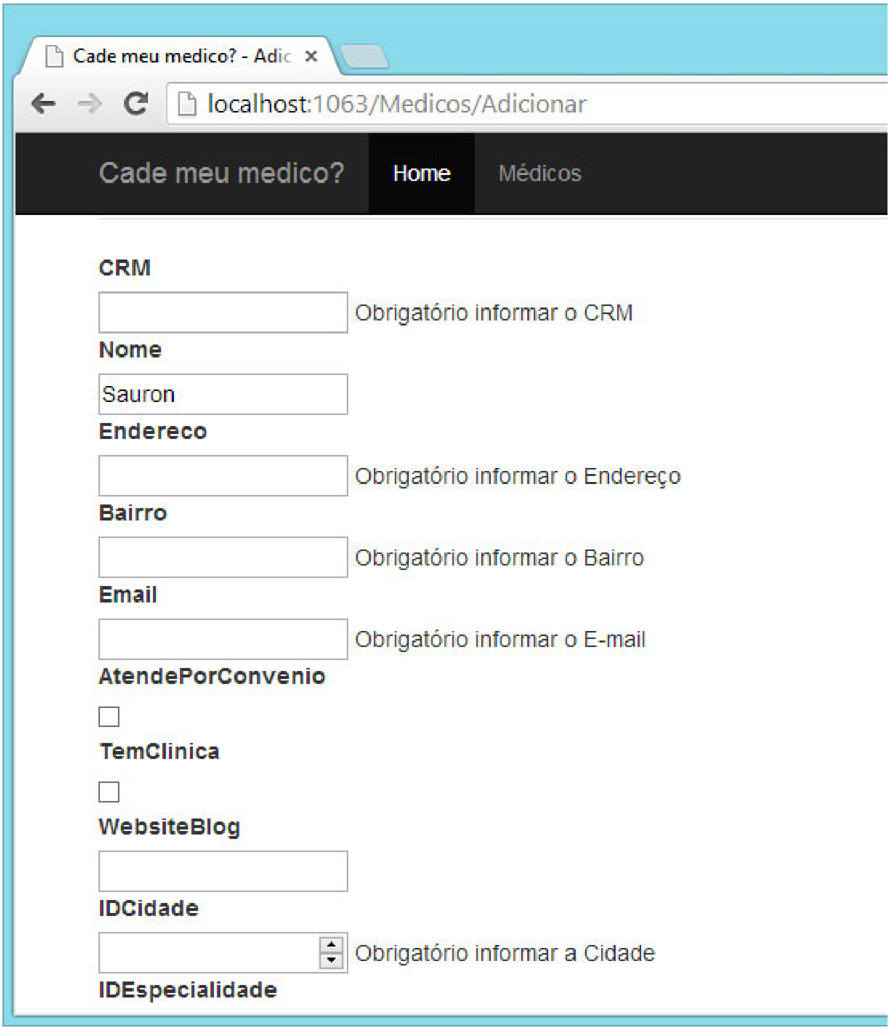


Figura 5.7: Validação ao adicionar médicos

O médico não será adicionado, já que campos obrigatórios não foram informados. Lembre-se que no capítulo sobre os *models* adicionamos atributos de validação nas propriedades das entidades. Nesse momento, os atributos estão sendo utilizados pelo MVC para fazer automaticamente a validação dos dados informados no formulário. Essa validação é feita pela linha "if (ModelState.IsValid)" da listagem 11.

No caso de um modelo válido, o *Entity Framework* adicionará o mesmo ao banco de dados. Caso algum problema seja encontrado na validação do modelo, a *view* será

apresentada novamente para o usuário com os dados anteriormente informados e com as mensagens da validação.

Informe os dados obrigatórios no formulário, clique no botão *Create* e, após a inclusão do registro, você será redirecionado para a *view* Index, na qual o médico recém-adicionado será apresentado.

Com a inclusão do médico concluída, é hora de avançar e fazer o formulário para a alteração dos cadastros. A listagem 12 demonstra o código das *actions* para a alteração dos médicos. Siga os passos apresentados anteriormente e crie a *View Editar*. Lembre-se de utilizar o template *Edit* na criação da *view* e fazer as alterações no HTML conforme apresentadas anteriormente. Uma diferença é que depois de excluir o campo IDMedico, o código `@Html.HiddenFor(model => model.IDMedico)` deve ser adicionado. A *view* deve possuir todos os dados do modelo em edição — no caso do IDMedico, o usuário não precisa ser exibido ao usuário, mas precisa estar presente no formulário.

Listagem 5.12 -

:

```
public ActionResult Edit(long id)
{
    Medico medico = db.Medicos.Find(id);

    ViewBag.IDCidade = new SelectList(db.Cidades, "IDCidade",
                                       "Nome",
                                       medico.IDCidade);
    ViewBag.IDEspecialidade = new SelectList(db.Especialidades,
                                              "IDEspecialidade",
                                              "Nome",
                                              medico.IDEspecialidade);

    return View(medico);
}

[HttpPost]
public ActionResult Editar(Medico medico)
{
    if (ModelState.IsValid)
    {
        db.Entry(medico).State = EntityState.Modified;
```

```
        db.SaveChanges();
        return RedirectToAction(%%Index%%);
    }
    ViewBag.IDCidade = new SelectList(db.Cidades, "IDCidade",
                                     "Nome",
                                     medico.IDCidade);
    ViewBag.IDEspecialidade = new SelectList(db.Especialidades,
                                            "IDEspecialidade",
                                            "Nome",
                                            medico.IDEspecialidade);
    return View(medico);
}
```

Após a criação das *actions* e da *view Editar*, só nos resta realizar mais uma alteração. Ela precisa ser feita na *view Index*. Cada linha da tabela que representa um médico cadastrado possui um link para sua Edição e Exclusão. Vamos primeiro alterar a criação do link para edição. Altere o código `@Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ })` para o código correto que direciona para a *action* `Editar`: `@Html.ActionLink("Editar", "Editar", new { id=item.IDMedico })`.

Feito isso, compile sua aplicação e teste a alteração do registro recém-criado.

Para finalizar as operações do cadastro de Médicos, só resta a criação da *action* para a exclusão dos registros. A listagem 13 representa o código da *action* `Excluir`. O código é mais simples do que os apresentados anteriormente: a *action* retorna true ou false de acordo com o resultado da exclusão. Esses valores serão utilizados no próximo capítulo, assim como a chamada para a *action* `Excluir` que será feita por Ajax no capítulo sobre *Views*.

Listagem 5.13 -

```
    :

[HttpPost]
public string Excluir(long id)
{
    try
    {
        Medico medico = db.Medicos.Find(id);
        db.Medicos.Remove(medico);
        db.SaveChanges();
```

```
        return Boolean.TrueString;
    }
    catch
    {
        return Boolean.FalseString;
    }
}
```

Vimos até aqui as operações para o cadastro de médicos. Utilize o que foi apresentado até o momento para criar também os cadastros de Cidades e Especialidades. Aproveite também para traduzir os elementos criados em inglês pelo MVC nas *Views* autogeradas. A partir de agora veremos com mais detalhes os recursos disponíveis nas *views* do MVC.

Capítulo 6

Views: interagindo com o usuário

No capítulo 4, você teve a oportunidade de se aprofundar no tema "*models*". Lá, você pôde entender um pouco mais sobre como o *framework* MVC trabalha com fontes de dados, principalmente utilizando o ORM da Microsoft — o *Entity Framework*. Assim, você pôde compreender a importância da letra "**M**" no acrônimo MVC.

Em seguida, você foi apresentado de forma mais íntima ao grupo de conceitos envolvidos na utilização dos *controllers*. Através de exemplos práticos e com explicações diretas em cima de cada linha de código, procuramos tornar evidente o "peso" imposto pela letra "**C**" no contexto de aplicações ASP.NET MVC.

Assim, após entender a forma com que podemos nos conectar aos bancos de dados e posteriormente impor comportamentos às aplicações, é chegado o momento em que poderemos entender de forma mais clara como o *framework* endereça o resultado final da integração entre as duas etapas mencionadas nos dois parágrafos anteriores, isto é, a exibição dos dados para o usuário final. É claro que estamos falando da letra M**V**C, que refere-se à palavra *Views* (em português brasileiro, "Visões").

6.1 Visões?!

Visões (ou como a grande maioria dos desenvolvedores prefere chamar — *views*) são os elementos que constituem a "ponta externa do *iceberg*" nas aplicações ASP.NET MVC. A associação é pertinente, uma vez que são os ponteiros das tais estruturas de gelo que podem ser vistas a olho nu sem que se quer possa imaginar a complexa estrutura que há por debaixo d'água. O mesmo ocorre com as *views* em aplicações ASP.NET MVC.

Como já mencionamos anteriormente neste livro, o principal objetivo do *framework* MVC é separar responsabilidades. Assim, temos basicamente três camadas, sendo que uma delas (as *views*) são dedicadas exclusivamente à exibição de informações ao usuário final. Não há processamento a nível de servidor nesta camada, já que, para isso, temos os *controllers*. De igual forma, acessos a dados não devem ser realizados nesta camada, uma vez que possuímos os *models* para tal. Assim, resta a *view* receber massas de dados já prontas e simplesmente exibi-las. A figura 6.1 apresenta o fluxo de operação até a *view*.

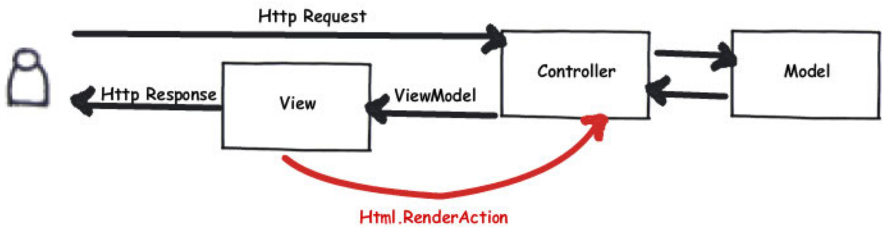


Figura 6.1: O fluxo de operação do framework MVC até a view. Fonte: http://www.arrangeactassert.com/

Levando-se em consideração os conceitos apresentados nos parágrafos iniciais desta seção, uma pergunta obrigatoriamente deve ser respondida: se a função das *views* é unicamente exibir dados já formatados, como podemos exibir tais informações? Nas seções seguintes, tentaremos responder principalmente a esta questão. Evidentemente, neste contexto, muitos novos conceitos serão apresentados.

6.2 Conceito de "Engenho de renderização"

As famosas *view engines* ou simplesmente "engenhos de renderização", são mecanismos presentes em todas as versões *framework* ASP.NET. Aqui chamamos de *view*

engine o mecanismo interno da plataforma (em nosso caso, ASP.NET) que possibilita a transformação de diretivas exclusivamente de servidor em código HTML, legível pelo navegador, ainda no *container* web para posterior exibição ao usuário final.

A ideia fundamental neste caso é a de proporcionar ao desenvolvedor o alto poder programático oferecido pelas linguagens de servidor — como as tags dos componentes de servidor do ASP.NET, por exemplo — e passar a responsabilidade de geração do código objeto no *browser* para o *container* web.

6.3 ASPX ou ASP.NET Razor?

Já mencionamos o fato de ASP.NET se tratar de uma plataforma e não uma linguagem de programação. Justamente por este motivo, todas as tecnologias agrupadas na plataforma recebem o prefixo "ASP.NET", como ASP.NET MVC.

Falando especificamente de *view engines*, atualmente encontram-se disponíveis na plataforma ASP.NET duas opções delas: ASP.NET (dos famosos arquivos com extensão **.ASPX*) e ASP.NET Razor (esta última, mais recente). Assim, ao criar um novo projeto web utilizando qualquer das tecnologias disponíveis na plataforma ASP.NET, o desenvolvedor poderá optar pela utilização daquela *view engine* que melhor atende às suas necessidades.

É evidente que cada uma das *view engines* mencionadas possui suas peculiaridades e características intrínsecas que a torna adequada aos diferentes tipos de projetos. Muito embora seja possível utilizar a *view engine* ASPX em projetos ASP.NET MVC, este não é o padrão recomendado pelo *framework*. Assim, como este livro mantém seu foco no estudo da versão 4 do *framework* MVC, e como para esta versão a *view engine* padrão é a ASP.NET Razor, nas próximas seções deste capítulo nos concentraremos na apresentação das principais características e recursos disponíveis neste motor de visualização. Por inércia, deverão ficar evidentes as funcionalidades que fizeram com que a Microsoft não apenas incorporasse o recurso ao *framework* como fizesse dele a *view engine* padrão.

6.4 ASP.NET Razor

A versão final do ASP.NET MVC 3 (entregue ao mercado em Janeiro de 2011) trouxe uma série de novos e importantes recursos. Você pode encontrar a relação de novidades trazidas pela versão através do link:

http://www.asp.net/mvc/mvc3#overview

Dentre estas novidades, aquela que recebeu maior destaque e que contribuiu em grande escala para a adoção do *framework*, foi a introdução da nova *view engine* — a ASP.NET Razor.

Diferentemente do que você pode estar pensando, ASP.NET *Razor* não é uma linguagem de programação. É, sim, uma nova forma de estruturar *views* que precisam de alguma porção de código processada no servidor de aplicação.

Costuma-se dizer que *Razor* é um modelo de escrita de código por não possuir uma linguagem predefinida. É possível utilizar o modelo de programação oferecido pelo *Razor* utilizando as duas principais linguagens da .NET *framework*: C# ou VB. No caso, arquivos *Razor* estruturados com base em C# possuem a extensão "*.cshtml", enquanto arquivos estruturados com base em VB possuem a extensão "*.vbhtml".

Como você já pôde perceber, em projetos ASP.NET MVC, as *views* possuem a função de exibir dados processados pelos *controllers*. Capturar valores provenientes destes dos *controllers* é uma tarefa amplamente simplificada pelo *Razor*.

6.5 Diferenças na prática

Para que possamos apresentar os conceitos chaves relacionados ao Razor que o diferenciam das demais abordagens, apresentaremos dois trechos de código, que realizam rigorosamente a mesma operação: repetir itens em uma lista fictícia via `foreach`. A listagem 1 apresenta o trecho de código escrito com ASPX.

Listagem 6.1 - Repetindo itens em uma lista com ASPX:

```
<% foreach(var item in itens) { %>
  <span>
    <%: item.ValorUnitario %>
  </span>
<% } %>
```

A listagem 2 a seguir apresenta um trecho de código que realiza a mesma operação realizada pelo código da 1, mas, utilizando *Razor*.

Listagem 6.2 - Repetindo itens em uma lista com Razor:

```
@foreach(var item in itens) {
  <span>
```

```
    @item.ValorUnitario
  </span>
}
```

Estes trechos de códigos são decisivos para apresentarmos algumas conceitos iniciais importantes acerca da *view engine Razor*. A primeira observação pertinente em relação às abordagens é a diferença que demarca os blocos de código. Enquanto *views* que utilizam ASPX utilizam delimitadores "<% %>", *Razor* utiliza apenas a marcação de "@". Outra observação (não menos importante) é que o motor de renderização do *Razor* é "inteligente". Perceba que, enquanto ASPX tem uma separação explícita entre suas tags e as HTML, *Razor* não a possui. Isso ocorre porque o motor de renderização do *Razor* separa de forma automática HTML de código C# (ou Visual Basic que também é suportado, é importante que se diga).

Evidentemente, esta não é a única forma de se escrever código *Razor*. Existem casos em que blocos de instruções devem ser agrupado em blocos maiores (multilinhas) para se conseguir uma melhor organização do código na *view*, por exemplo. Para estes casos, poderíamos utilizar uma abordagem conhecida como "Bloco de múltiplas linhas". Caso uma linha seja suficiente para expressar um comportamento através do *Razor*, o método conhecido como "inline" pode ser utilizado. A listagem 3 apresenta a diferença entre as abordagens.

Listagem 6.3 - Multilinhas versus Inline:

```
@{
  //Multi-linhas
    ViewBag.Title = "Teste de layout";
    var Data = DateTime.Now.DayOfWeek;
    string StringConcatenada = "Hoje é '" + Data.ToString()
        + "'. Seja bem vindo(a)!";
}

<!-- Inline -->
<h2>@StringConcatenada</h2>
```

Como você pôde perceber, com a utilização do *Razor* é possível declarar variáveis e instanciar objetos, e muito mais. É possível também, conforme vimos na listagem 2, criar estruturas de repetição que permitem ao desenvolvedor navegar entre estruturas de dados simples (*arrays*, por exemplo) ou complexas (dicionários de dados, listas multi-valoradas etc.). Se toda a estrutura C#/VB é suportada na *view*

que utiliza *Razor*, estruturas de tomadas de decisão (falamos if/else, switch/case e suas variações) também as são — ver listagem 4. Tudo isso utilizando não apenas o modelo de programação orientado a objetos, já familiar para quem trabalha com .NET, mas principalmente, utilizando a mesma sintaxe.

Listagem 6.4 - Verificando valor de entrada na view com if/else:

```
@{
    ViewBag.Title = "Teste de layout";
    var Data = DateTime.Now.DayOfWeek;
    string StringConcatenada = Data.ToString();
}

<!-- Verificando o dia retornado -->
@if (StringConcatenada == "Friday") {
    <h2>Sexta-feira</h2>
}
else
{
    <h2>Outro dia qualquer da semana.</h2>
}
```

Trabalhar com ASP.NET Razor é simples e não exige do desenvolvedor (*designer*) conhecimentos adicionais. Basta que se entenda o modelo de trabalho. Nas próximas seções apresentaremos alguns outros aspectos funcionais disponibilizados pela tecnologia. Você irá adorar!

6.6 Helpers?!

Não há dúvidas entre aqueles que utilizam *Razor* em seu dia a dia quanto à utilidade dos *helpers* no contexto das aplicações.

A criação de *helpers* é um recurso que se encontra disponível no contexto do ASP.NET *Razor* desde a primeira versão. Inicialmente, *Razor* permitia que os *helpers* apenas retornassem *strings* para as *views*. Felizmente, esta realidade mudou e, hoje, podemos criar *helpers* que retornam porções e HTML com objetos atrelados. Isso dá ao desenvolvedor maior poder e, consequentemente, maior robustez às *views*.

De forma direta, são os *helpers* que permitem criar operações e comportamentos específicos que poderão ser reutilizados ao longo de toda a aplicação. Considere

novamente nosso projeto "*Cadê meu médico*". Imagine, por exemplo, que queremos exibir ao longo de toda a aplicação um *box* de publicidade, onde exibiremos os dois *banners* mais recentes. Com o objetivo de desacoplar os componentes da *view* e reutilizar em outras aplicações, inclusive, poderíamos criar um *helper* "BannersPublicitarios".

Antes de criarmos nosso *helper* de exemplo, é importante saber que existem dois modelos para a criação de *helpers*. A saber, na própria *view* onde ele será utilizado ou de forma que ele possa ser reaproveitado ao longo de múltiplas *views*. Como o objetivo principal dos *helpers* é servir como mecanismo de *refactoring* de códigos na *view*, apresentaremos no exemplo a seguir apenas a segunda abordagem.

Criando o helper "BannersPublicitarios"

Com o projeto "*Cadê meu médico*" em execução, vá até a *Solution Explorer* e sobre o nome do projeto, clique com o botão direito. Selecione as opções `Add > Add ASP.NET Folder > App_Code`. Claro, você deverá realizar este procedimento apenas se o diretório mencionado não existir em sua *Solution Explorer*. O diretório "App_Code" é outra das convenções (CoC) implementadas pelo *framework* ASP.NET MVC para diretórios. Ao adicionar uma classe ou arquivo *Razor*, o projeto MVC entenderá que o referido arquivo trata-se de um script que poderá ser reutilizado em outras partes da aplicação como um código agregado de função específica (como é o caso dos *helpers*).

No passo seguinte, sobre o diretório "App_Code", clique com o botão direito e selecione a sequência de opções: `Add > New item`. Na janela que se abrir, selecione a opção *MVC 4 View Page (Razor)* e a nomeie como "BannersPublicitarios.cshtml", conforme apresenta a figura 6.2.

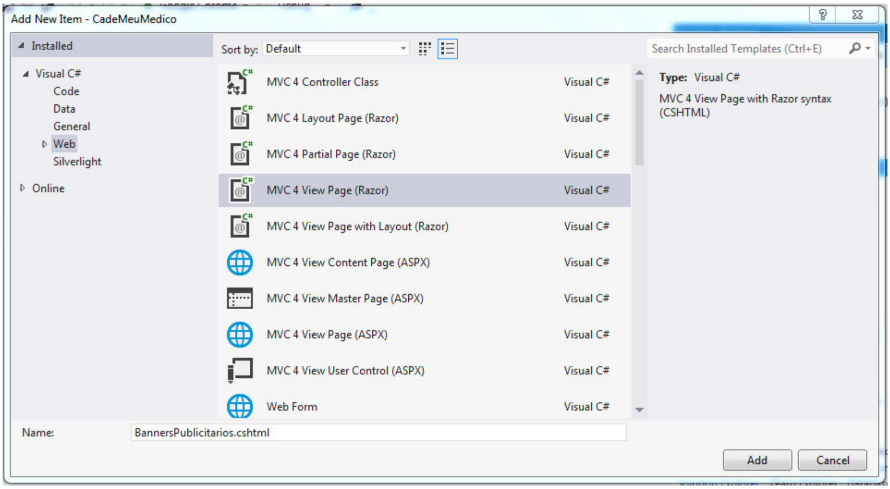


Figura 6.2: Adicionando o arquivo do helper BannersPublicitarios

Antes de irmos aos códigos, relembremos: o que faremos aqui é um painel que exibe *banners* publicitários. Para que isto possa ser dinâmico, adicionarei uma nova tabela ao nosso banco de dados chamada "BannersPublicitarios" e a preencherei com dois registros: um para "Conio Soluções em Tecnologia" com sua respectiva imagem e outro "Casa do Código" também com sua imagem. Só então partiremos para a codificação de nosso *helper*.

O SQL para a criação da tabela e os respectivos *insert's* são apresentados pela listagem 5.

Listagem 6.5 - Script para criação da tabela BannersPublicitarios e seus respectivos inserts:

```
CREATE TABLE BannersPublicitarios
(
  IDBanner BIGINT IDENTITY NOT NULL,
  TituloCampanha VARCHAR(60) NOT NULL,
  BannerCampanha VARCHAR(200) NOT NULL,
  LinkBanner VARCHAR(200) NULL,

  PRIMARY KEY(IDBanner)
);
```

```
INSERT INTO BannersPublicitarios
(TituloCampanha, BannerCampanha, LinkBanner) VALUES
('Campanha Conio', 'logo-conio-cademeumedico.png','http://conio.com.br')

INSERT INTO BannersPublicitarios
(TituloCampanha, BannerCampanha, LinkBanner) VALUES
('Campanha Casa do Código', 'banner-cdc-cademeumedico.png',
'http://casadocodigo.com.br')
```

Os *banners* que aparecem nos dois *inserts* encontram-se no projeto completo no GitHub e você pode utilizá-los para completar o exemplo.

Agora que possuímos nossa infra de banco de dados montada, podemos voltar ao Visual Studio para implementar, de fato, o *helper* que exibirá estas informações na aplicação. Após atualizar o modelo da aplicação (sim, isso é preciso para que o *Entity Framework* "enxergue" a nova tabela), abra o arquivo "BannersPublicitarios.cshtml" localizado em "App_Code". Substitua o conteúdo gerado de forma automática pelo Visual Studio pelo apresentado pela listagem 6.

Listagem 6.6 - Action que retorna em ViewBags os banners publicitários:

```
@using CadeMeuMedico.Models;

@helper RetornaDoisBannersMaisRecentes() {

var bd = new CadeMeuMedico.Models.EntidadesCadeMeuMedicoBD();
var banners = bd.BannersPublicitarios.
      OrderByDescending(b => b.IDBanner).Take(2);

<div style="width:100%; text-align:left; border:1px
    solid #efefef; padding:10px; display:inline-block;">

    @foreach(var b in banners) {
    <div style="width:125px; height:125px;
                float:left; margin-right:10px;">
        <a href="@b.LinkBanner">
            <img src="~/Uploads/Banners/@b.BannerCampanha"
                                        title="@b.TituloCampanha" />
        </a>
    </div>
    }
```

```
</div>

}
```

Alguns pontos importantes relacionados ao código apresentado pela listagem 6:

- Utilizamos a diretiva *using* direto da *view*. Lembre-se, aqui é programação C# da forma como você já conhece. O que *Razor* faz é possibilitar tal mecanismo em uma *view*. Nunca é demais lembrar;
- Através da diretiva "@helper" informamos ao *framework* que o trecho de código a seguir deverá se comportar como um *helper*;
- Na sequência, crio uma instância do modelo de dados, busco os dois banners mais recentes e exibo já na estrutura HTML com *Razor*.

Isso posto, resta-nos apresentar a forma através da qual invocamos o *helper* em nossa aplicação. Antes de apresentarmos o código, entretanto, é preciso entender que, uma vez criado da forma correta, o *helper* poderá ser invocado em qualquer parte da *view* da aplicação (*views* comuns ou *master pages*). O código que invoca o *helper* recém-criado pode ser visualizado através da listagem 7. Para realizar a demonstração, estamos invocando o *helper* "BannersPublicitários" na *view* "Index" do *controller* "Home".

Listagem 6.7 - Invocando o helper de banners publicitários:

```
@BannersPublicitarios.RetornaDoisBannersMaisRecentes();
```

Note que a chamada é simples de ser realizada. Basta informar o nome do arquivo *helper* (no caso "BannersPublicitarios") e, na sequência, invocar o método "RetornaDoisBannersMaisRecentes()". Isso automaticamente nos diz que um *helper* pode implementar vários métodos. Em nosso exemplo, poderíamos ter tranquilamente, um segundo método chamado "RetornaSeisBannersMaisRecentes()" etc. O resultado gerado pela implementação deste *helper* pode ser visualizado através da figura 6.3.



Figura 6.3: Helper incorporado a view principal da aplicação exemplo

Uma observação se faz necessária neste ponto, antes de prosseguirmos: Antes que você nos crucifique por estarmos realizando um acesso ao banco de dados diretamente da *view*, é importante entender que tal procedimento foi realizado apenas para simplificar o processo, facilitando o entendimento do conceito que desejamos passar. Vale observar aqui que o modelo ideal seria retornar os dados através de um *controller* e, aí sim, com os dados já disponíveis, navegar entre eles através da *view* utilizando *Razor*.

Helpers nativos

Ainda sobre os *helpers*, é importante notar o seguinte: existem aqueles personalizados (criados sob demanda, a exemplo do que fizemos no tópico anterior) e existem os helpers nativos do *framework*, que podem ser utilizados para otimizar algumas operações com *views*. Você já está em contato com eles desde o início deste livro e talvez ainda não saiba. Para exemplificar sua utilização e importância, vamos considerar novamente a listagem 5.

Se você é um bom observador, por certo percebeu que o formulário foi gerado basicamente através de um componente "@Html". @Html

é um *helper* nativo que encapsula a criação de componentes HTML através de código com *Razor*. O que queremos dizer com isso é que, quando você escreve a linha `@Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ })`, por exemplo, você está dizendo ao motor de renderização do *Razor* que ele deverá realizar um *render* para a tag `<a>` do HTML em tempo de execução.

Muito embora você não precise necessariamente destes *helpers* para construir suas *views* (sim, você pode utilizar HTML puro e utilizar *Razor* apenas para pegar as informações provenientes do *controller*), ao utilizar os *helpers Razor* é possível obter ganhos principalmente em duas situações:

- **Quando mudanças estruturais ocorrerem**: para muitos tipos de projetos, o modelo de rotas padrão não endereça as demandas. Assim, ao adicionar uma rota personalizada para sua aplicação, por exemplo, no caso de a *view* estar estruturada com *helpers* como o @Html, de forma automática o *framework* é capaz de ajustar os links das aplicações.

- **Linguagem mais familiar para desenvolvedores escreverem *views***: HTML é, de forma geral, mais natural para *designers* do que para *desenvolvedores*. Como a nomenclatura para a utilização dos *helpers* é mais próxima do C#, o processo de escrita de *views* para desenvolvedores acaba sendo mais suave.

De forma geral, o preço a se pagar pelo uso excessivo de *helpers* nas *views* é a singela perda de desempenho, uma vez que, ao fazê-lo, colocamos uma camada a mais de processamento sobre o processo de renderização (lembre-se, *Razor* é processado no servidor).

Este tipo de *helper* pode ainda ser estendido (sim, você pode criar suas extensões) através dos *extension methods*, recurso disponível no .NET *framework* desde a versão 3.5. Você pode encontrar uma boa leitura acerca disso através deste link (em inglês): http://bit.ly/15Zr9cs.

Vale mencionar, ainda, que existe uma grande quantidade de *helpers* já prontos, que realizam diferentes tarefas, disponíveis de forma gratuita no repositório público do NuGet. Dentre os mais baixados, podemos citar: integração com Twitter (a.k.a., Twitter.Goodies), Facebook, PayPal e Amazon. O processo de instalação destes *helpers* já é conhecido por você, uma vez que segue o mesmo padrão imposto pelo "Library Package Manager" (NuGet) do Visual Studio, apresentado nos capítulos iniciais do livro.

Para invocá-lo, você deverá seguir o mesmo padrão apresentado por nós no exemplo da criação e utilização do *helper* "BannersPublicitarios". Aliás, este é um bom momento para treinar! Vá até o repositório do NuGet através do Visual Studio, busque pelo helper "Twitter.Goodies" e instale-o. Isso feito, adicione-o a uma *view* qualquer do projeto de exemplo e veja o resultado "mágico" da integração.

6.7 Algumas outras vantagens

Evidentemente, utilizar uma *view engine* proposta pela Microsoft tem suas vantagens (seja ASPX ou *Razor*). Existem recursos disponíveis nelas que não recebem grande atenção por parte da maioria dos desenvolvedores mas que, se utilizados da forma correta, podem acarretar em ganho de produtividade, melhorar a qualidade do código e, ainda, melhorar a performance das páginas. A seguir discutiremos alguns destes aspectos.

Recursos para explicitar caminhos (path's)

Se você já trabalhou com web em qualquer nível (e nós imaginamos que sim), com certeza já constatou que determinar caminhos de arquivos para as aplicações não é lá das coisas mais simples. Isso ocorre por alguns fatores. Por exemplo, determinados provedores de *host* para aplicações web bloqueiam a utilização do recurso de *parent path* por questões de segurança. Dessa forma, os caminhos devem ser informados de forma absoluta e não relativa. O problema neste caso é que como a aplicação é desenvolvida localmente e os caminhos locais podem ser diferentes daqueles utilizados pelo servidor web, ao realizar a publicação da aplicação no ambiente remoto, é comum encontrar a ocorrência de páginas com referências quebradas. Em aplicações grandes, isso acaba tornando-se um problema complexo de ser resolvido.

Além disso, utilizar caminhos relativos em aplicações de grande porte que possuem muitos diretórios aninhados pode ser uma tarefa complexa, que toma muito tempo de manutenção do código.

Reconhecendo esta dificuldade, a Microsoft propôs um modelo extremamente funcional e elegante para realizar esta tarefa em suas *view engines*. Para que uma referência a determinado arquivo possa ser realizada, basta adicionar o símbolo "~" (til) no início do caminho absoluto. Desta forma, para um caminho absoluto "/Uploads/Imagens/" teríamos uma nova especificação deste mesmo caminho da seguinte forma: "~/Uploads/Imagens". Ao assim fazer, estamos "dizendo" ao IIS que, independente do diretório atual, ele deve deve buscar o recurso informado a partir da

raiz até o ponto especificado. Simples, não?!

Renderizações parciais

Quando iniciamos a criação da aplicação "*Cadê meu médico?*", como você deve se lembrar, uma de nossas primeiras providências foi a de criar a página mestra, conhecida entre os programadores/designers como *master page*. Esta página foi criada com o objetivo de servir de padrão visual para todas as demais *views*, que carregariam seus conteúdos dentro de uma área especifica da *master page*. A diretiva que possibilitou a realização desta mágica foi a "@RenderBody()". Certo?!

No sentido de desacoplar ainda mais a *master page* (você se lembra da história do "dividir pra conquistar"?), encontra-se disponível nas *view engines* da Microsoft um recurso conhecido como "renderizações parciais".

A ideia com este recurso é que você renderize porções específicas de código em pontos distintos da *master page*. Imagine, por exemplo, que seja importante que o menu da aplicação seja carregado de forma assíncrona enquanto o restante da *view* seja carregado no primeiro *post*.

Você poderia utilizar a diretiva "@RenderSection()" ou "@RenderPartial()" para buscar uma *view parcial* que carrega (de forma assíncrona com jQuery, por exemplo) seu menu. A utilização de "@RenderPartial()" já foi demonstrada neste livro, no capítulo 3. A diferença básica de "@RenderPartial" para "@RenderSection" é que a primeira tem o poder de renderizar uma *view* parcial completa, e a segunda renderiza seções especificas. Para "@RenderPartial", você deverá especificar a *view* parcial que será renderizada naquele ponto.

Agregação e minificação

Nos dias atuais é comum que uma aplicação web possua milhares de linhas de código JavaScript, CSS, XML e afins. Diariamente nos cercamos de *plugins* para otimizar tarefas e proporcionar ao usuário final maior responsividade.

É claro que, quanto maior é a quantidade de código desta natureza em uma aplicação, invariavelmente, mais comprometida estará a performance das páginas. Desta forma, novamente a Microsoft propôs um recurso interessante para minimizar este problema: trata-se da agregação e minificação.

Já que para realizar a renderização os motores realizam a verificação do arquivo fonte, linha a linha, a ideia básica é que o próprio servidor de aplicação seja capaz de realizar, em tempo de execução, a remoção daquilo que pode ser desprezado sem

comprometer o resultado final. Estamos falando de espaços em branco, comentários e união das linhas (minificação), assim como, a junção em um único arquivo em memória (*bundle*) de todos os recursos deste tipo (*agregação*).

Para que esta ideia torne-se mais clara, nada melhor que um exemplo, não é mesmo? O exemplo que apresentarei a seguir foi retirado do blog oficial do Scott Guthrie e está disponível também através do link: http://bit.ly/19I5ODr.

Imagine que, em seu projeto, você possui a *solution explorer* com uma série de arquivos CSS (conforme ilustra a figura 6.4), todos eles utilizados por sua aplicação.

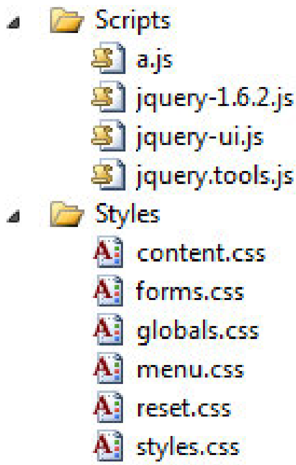


Figura 6.4: Solution explorer apresenta os arquivos CSS

No modelo de *web page* tradicional, para que estes arquivos possam entrar em atividade é preciso que haja uma referência manual para cada um deles, como ilustrado através da listagem 8.

Listagem 6.8 - Referenciando os arquivos de forma manual em uma view:

```
<link href="~/styles/reset.css" rel="Stylesheet" />
<link href="~/styles/styles.css" rel="Stylesheet" />
<link href="~/styles/content.css" rel="Stylesheet" />
<link href="~/styles/globals.css" rel="Stylesheet" />
<link href="~/styles/forms.css" rel="Stylesheet" />
<link href="~/styles/menu.css" rel="Stylesheet" />
```

Muito embora esta abordagem funcione perfeitamente, o problema evidente está

no fato de que, sequencialmente, são realizadas seis requisições ao servidor de aplicação. Não é preciso pensar muito para entender que milhões de acessos — quando falamos em performance é sempre bom imaginar se a estrutura atual se comportaria bem com milhões de acessos — requisitando seis arquivos todas as vezes é bem pior que uma requisição, certo?

Neste sentido, ao construirmos *views* com *Razor* ou ASPX podemos simplesmente fazer da seguinte forma (listagem 9).

Listagem 6.9 - Modelo de referencia contemplando agregação e minificação:

```
<link href="~/styles" rel="Stylesheet" />
```

Quando a *view Razor* ou ASPX encontrar este modelo de referência, o que ela (*framework*) fará é verificar todos os arquivos CSS dentro do diretório apontado (neste caso "styles"), combiná-los, minificá-los e, na sequência, devolver uma resposta HTTP de um único arquivo CSS. Assim, ao invés de seis chamadas serem realizadas junto ao servidor, este número cai para apenas uma chamada. Legal, não?!

6.8 Mobilidade: Sua consulta de médicos em dispositivos móveis

No início deste livro, falamos um pouco sobre algumas das principais necessidades da web atual. Como você deve se lembrar, dentre os principais aspectos mencionados naquela ocasião, encontrava-se a "mobilidade". De fato, aplicações que adaptam seus comportamentos de acordo com o dispositivo merecem um pontinho a mais no quesito qualidade, haja vista a popularização dos *smartphones* e *tablets*.

Ao estruturar aplicações utilizando ASP.NET MVC (a partir da versão 4) você pode utilizar recursos nativos para a construção de *views* que respondam de forma adequada para dispositivos móveis. Além disso, se você utiliza um *framework* como aquele que estamos utilizando (Twitter Bootstrap) para a estruturação visual da aplicação exemplo "*Cadê meu médico?*", poderá se servir também de forma nativa, deste tipo de recurso. Desta forma, apresentaremos a seguir as linhas gerais para que possamos estruturar nossas aplicações para funcionarem de forma adequada independente do dispositivo executor. Claro, utilizaremos a aplicação "*Cadê meu médico?*" como base para isso.

Bootstrap como plataforma de mobilidade

Já apresentamos formalmente há alguns capítulos (quando criamos nossa primeira aplicação MVC) o Twitter Bootstrap. Como este *framework* foi escolhido por nós para nos ajudar a compor os elementos visuais da aplicação "Cadê meu médico?", faz todo sentido falar sobre o seu aspecto *mobile*.

Atualmente, o Twitter Bootstrap encontra-se na terceira versão. Muito embora hoje o *framework* ofereça suporte completo e irrestrito à mobilidade, nem sempre foi assim. A primeira versão em nada contemplava mobilidade. Na segunda versão, alguns componentes já suportavam a autoadequação. Hoje, como o próprio time do Bootstrap gosta de afirmar, "A mobilidade vem primeiro. (*Bootstrap is mobile first.*)".

A grande pergunta aqui é: como é possível fazer com que uma *view* se ajuste de forma automática à tela do dispositivo com Bootstrap? A resposta é objetiva e extremamente simples: adicione uma *meta tag* com os parâmetros corretos. Ao fazê-lo, o CSS padrão do Bootstrap encaminhará o estilo adequado ao dispositivo que está requisitando o acesso e fará o ajuste para sua tela. Um exemplo da *meta tag* à qual nos referimos pode ser visualizada na listagem 10.

Listagem 6.10 - Meta tag que indica a mobilidade:

```
<meta name="viewport" content="width=device-width,
initial-scale=1.0, maximum-scale=1.0, user-scalable=no">
```

Em linhas gerais, estamos dizendo ao navegador o seguinte: "Por favor, execute o CSS adequado com base neste *viewport* e aplique os parâmetros explicitados no atributo *content*." Não entraremos em maiores detalhes acerca de cada uma destas propriedades e suas respectivas funções, já que este não é o objetivo do livro. Entretanto, você poderá encontrar excelentes leituras acerca destes assuntos através dos seguintes links:

- Manipulando a *meta tag* viewport: http://bit.ly/metatagviewport
- CSS Bootstrap (em inglês): http://bit.ly/cssbootstrap
- CSS Media Query: http://bit.ly/cssmediaquery

Mobilidade nativa com ASP.NET MVC

Como já mencionamos, o projeto "Cadê meu médico?" está estruturado sobre o *framework* Twitter Bootstrap, portanto, naturalmente, ele já se encontra autoajustável para o dispositivo utilizado pelo usuário (veja a figura 6.5). Entretanto, o *framework* ASP.NET MVC oferece suporte nativo para mobilidade de aplicações web. A seguir, discutiremos os possíveis formatos de trabalho para que isso ocorra.



Figura 6.5: Projeto 'Cade meu médico?' em funcionamento no emulador de iPhone

Quando falamos em mobilidade com ASP.NET MVC, podemos imaginar ao menos três abordagens distintas, passíveis de implementação. Estas são apresentadas resumidamente na lista a seguir:

- **Utilizar os mesmos *controllers* e *views* com diferentes *layouts Razor* dependendo do dispositivo do usuário**. A ideia aqui é que você utilize a mesma lógica implementada nos *controllers* e *actions*, assim como as mesmas *views*.

Neste caso, seriam renderizados em tempo de execução apenas os *layouts*. Esta abordagem é recomendada, de forma geral, quando se deseja apenas exibir dados (grids, por exemplo) de forma personalizada, ajustada ao dispositivo;

- **Utilizar os mesmos *controllers*, entretanto, renderizando *views* específicas, de acordo com o dispositivo**. Neste caso, ao invés de renderizarmos *layouts* personalizados, renderizamos *views* específicas. Esta opção deve ser sua escolha se você identificar a necessidade de modificar muito o HTML para os diferentes dispositivos. Outro aspecto importante a ser observado é se é preciso manter o fluxo de operações da aplicação ao longo dos dispositivos;
- **Criar áreas separadas para projetos *mobile* e projetos *desktop***. Este é o modelo onde a separação de camadas é mais definitiva. Existem diferentes projetos, *mobile* e *desktop* (com diferentes *controllers*, *actions* e *views*) em uma mesma solução. Ao receber a solicitação, o *framework* identifica a resolução do dispositivo e direciona o fluxo de operação para o projeto adequado na solução.

Para que esta ideia possa ficar mais clara, vamos a alguns exemplos práticos. Imagine que para a aplicação "Cadê meu médico?", escolhêssemos a primeira abordagem apresentada pela lista anterior. Para que a aplicação seja capaz de renderizar o *layout* correto (claro, imaginando que exista um *layout* chamado "_LayoutMobile.cshtml") bastaria escrever o código apresentado pela listagem 11 no interior do arquivo "_ViewStart.cshtml".

Listagem 6.11 - Verificando o layout a ser renderizado:

```
@{
    Layout = Request.Browser.IsMobileDevice ?
    "~/Views/Shared/_LayoutMobile.cshtml"
    : "~/Views/Shared/_Layout.cshtml";
}
```

O código é simples e dispensa comentários detalhados. Basta dizer que estamos verificando a resolução do navegador. Caso seja *mobile*, "solicitamos" ao *framework* que considere o *layout* intitulado "_LayoutMobile.cshtml".

Se optássemos pela segunda abordagem, poderíamos criar uma sobrecarga do método "FindView" (de "ViewEngineResult") para encontrar a *view* adequada, conforme apresenta a listagem 12.

Listagem 6.12 - Encontrando views específicas (sugestão de Scott Hanselman em seu blog):

```
public class MobileCapableWebFormViewEngine : WebFormViewEngine
{
    public override ViewEngineResult FindView(
      ControllerContext controllerContext,
      string viewName, string masterName, bool useCache)
    {
        ViewEngineResult result = null;
        var request = controllerContext.HttpContext.Request;

        if (request.Browser.IsMobileDevice)
        {
            result = base.FindView(controllerContext, "Mobile/" +
                     viewName, masterName, useCache);
        }

        if (result == null || result.View == null)
        {
            result = base.FindView(controllerContext, viewName,
                                   masterName, useCache);
        }

        return result;
    }
}
```

É claro que o código acima apresenta apenas uma ideia do que se pode fazer. Este tipo de operação pode ser implementada, por exemplo, em um *controller* base de sua aplicação (que herde de *controller*).

Escolhendo a terceira opção, poderíamos criar uma área específica para uma aplicação móvel dentro de nosso projeto. Para isso, bastaria clicar com o botão direito sobre ele e clicar na opção `Add > Area` (nos moldes do que já fizemos durante todo o livro).

No interior dessa área podemos adicionar *controllers* e *views* normalmente, como se fosse um projeto à parte. Assim, para fins didáticos, podemos adicionar um *controller* "HomeController" dentro da nova área. Este *controller* poderá atuar como o novo controlador padrão da aplicação em dispositivos móveis. Bastaria,

após isso, realizar dois pequenos ajustes no padrão rotas da aplicação (através dos arquivos "RouteConfig.cs"), para garantir que tal *controller* poderá ser alcançado. Estes ajustes podem ser visualizados na listagem 13.

Listagem 6.13 - Ajustando a rota padrão da aplicação:

```
//Alterando a rota padrão...

public override void RegisterArea(AreaRegistrationContext context)
{
    context.MapRoute(
        "Mobile",
        "Mobile/{controller}/{action}/{id}",
        new { controller = "Home", action = "Index",
              id = UrlParameter.Optional }
    );
}

//Dando prioridade para o controller de desktop

public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        "Default",
        "{controller}/{action}/{id}",
        new { controller = "Home", action = "Index",
              id = UrlParameter.Optional },
        // Adicione o namespace dos controladores desktop a seguir
        new[] { "CadeMeuMedico.Controllers" }
    );
}
```

Na sequência, para coroar o trabalho, bastaria criar um filtro que redirecionaria o usuário para o conteúdo *mobile*, se isto for pertinente, claro. Como você deve se lembrar do início deste livro, estamos trabalhando o conceito de personalização de rotas, com o fim de definir as prioridades de chamadas e execuções, uma vez que possuímos dois *controllers* "Home" em nossa aplicação.

Concluindo e desafiando

Neste capítulo apresentamos todos os principais conceitos para que você possa ampliar seus conhecimentos em relação à maneira com que o *framework* MVC trata o mecanismo de *views*. Apresentamos também importantes conceitos acerca do ASP.NET *Razor*, *view engine* padrão da Microsoft para projetos web. Finalmente concluímos apresentando conceitos de mobilidade dentro do ASP.NET MVC.

Fica como desafio para este capítulo reproduzir na prática qualquer um dos três mecanismos de mobilidade nativos do ASP.NET MVC mencionados aqui. Dica: nós recomendamos o terceiro modelo.

Capítulo 7

Segurança: Criando sua área administrativa

Quando elencamos as principais necessidades da web nos dias atuais (e você deve se lembrar do início deste livro), "segurança" é um item que aparece no topo da lista e, claro, não poderia ser diferente. Assegurar ao usuário de determinado sistema que o acesso a seus dados e recursos estão protegidos, deixou de ser apenas um diferencial competitivo, mas sim, é uma obrigação.

Especialmente na web, segurança é de fato um dos aspectos mais importantes de uma aplicação, haja vista a natureza do ambiente: um lugar público.

ASP.NET MVC oferece uma ampla gama de recursos que podem lhe ajudar a criar aplicações realmente seguras. Evidentemente, não será possível apresentar em detalhes, todos estes recursos, entretanto, você será guiado pelo processo de construção de uma área de acesso restrito que utiliza claro, os principais recursos disponíveis no *framework* MVC para este fim. Vamos começar?!.

7.1 Área administrativa?!

Como você deve se lembrar, a aplicação "Cade Meu Medico?" se divide em duas frentes: uma interface pública (website) acessada pelo usuário final para realizar a consulta do médico desejado; uma área restrita através da qual é realizada a administração deste website.

Até aqui, várias rotinas para a área administrativa (como o cadastro de médicos apresentado no capítulo 5) foram criadas mas, da forma como estão, elas podem ser acessadas por qualquer usuário. Precisamos agora adicionar "muros" e "portões" a esta aplicação de forma a permitir apenas a entrada de pessoas autorizadas.

7.2 Criando a área administrativa

Iniciaremos nossas atividades criando a "porta de entrada" da área administrativa de nossa aplicação, já que suas subáreas serão protegidas por usuário e senha.

Existem diferentes maneiras para se construir mecanismos de autenticação e persistir os dados do usuário autenticado utilizando ASP.NET MVC. Poderíamos utilizar o mecanismo de autenticação do próprio *ASP.NET Membership* (você pode encontrar um bom texto sobre o assunto no link: http://bit.ly/aspnetmembership) , poderíamos criar nosso próprio modelo de autenticação utilizando o já super difundido conceito de *sessions* (você pode ler um bom texto sobre esta abordagem através do link: http://bit.ly/aspnetsessions) , poderíamos utilizar recursos de cache, tabela de um banco de dados relacional, enfim.

Para a construção de nossa aplicação, utilizaremos um recurso amplamente utilizado *web* afora: os *cookies*. Se este conceito é novo para você, recomendamos a leitura do texto contido neste link: http://bit.ly/conceitodecookies.

Execute o Visual Studio. Em seguida, navegue até a *Solution Explorer* e na raiz do projeto, adicione um novo diretório. Nomeie-o como "Repositorios". Em seu interior, adicione uma nova classe e atribua a ela o nome "RepositorioUsuarios.cs". Em seu interior, adicione o método apresentado pela listagem 1.

Listagem 7.1 - Adicionando o método que verifica a existência do usuário:

```
public static bool AutenticarUsuario(string Login, string Senha)
 {
     var SenhaCriptografada =
        FormsAuthentication.
```

```
        HashPasswordForStoringInConfigFile(Senha, "sha1");
     try
     {
        using (EntidadesCadeMeuMedicoBD db =
                new EntidadesCadeMeuMedicoBD())
        {
            var QueryAutenticaUsuarios =
                db.Usuarios.
                Where(x => x.Login == Login && x.Senha == Senha).
                SingleOrDefault();

            if (QueryAutenticaUsuarios == null)
            {
                return false;
            }
            else
            {
                RepositorioCookies.RegistraCookieAutenticacao(
                QueryAutenticaUsuarios.IDUsuario);
                return true;
            }
        }
    }
    catch (Exception)
    {
        return false;
    }
}
```

O código apresentado pela listagem 1 é simples mas apresenta alguns aspectos importantes. Ao recebermos os dados de "Login" e "Senha", verificamos através de uma expressão lambda — um bom post sobre este assunto pode ser encontrado através deste link: http://bit.ly/expressoeslambda — se existe na base de dados algum registro que atende de forma única aos dados informados pelo usuário. Se encontrar, gravamos o código do usuário em um *cookie* e o registramos no navegador e retornamos o valor *true* para indicar o sucesso da operação, caso contrário, retornamos o valor *false* para indicar a falha.

Duas observações importantes aqui. A primeira diz respeito à linha " `var SenhaCriptografada = FormsAuthentication.HashPasswordForStoringInConfigFile(Senha,`

`"sha1");`". Estamos fazendo desta forma por imaginar que a senha de acesso do usuário encontra-se criptografada no banco de dados — sim, esta é uma boa prática. Desta forma, estamos realizando a verificação da senha já criptografada. A segunda observação está relacionada à utilização de repositórios para realizar a tarefa de gravação/registro do *cookie* ("`RepositorioCookies.RegistraCookieAutenticacao( QueryAutenticaUsuarios.IDUsuario);`").

Repositórios? O que é isso?

Um novo conceito importante está presente propositalmente no código apresentado pela listagem 1 — **Repositórios**. Note, os repositórios agrupam em classes específicas, métodos que executam ações de mesma natureza. Em nosso exemplo, para agrupar todas as operações relacionadas a "Usuarios", criamos uma classe de repositório chamada "RepositorioUsuarios". Para facilitar a manutenção de código, criamos um diretório em nosso projeto chamado "Repositorios", que reúne todas as classes de repositório do projeto.

Repositórios são amplamente utilizados em projetos de *softwares*, uma vez que ajudam a melhorar o *design* (entenda-se, arquitetura) dos mesmos.

Outra observação importante em relação à utilização dos repositórios é a necessidade de declaração explícita de "CadeMeuMedico.Repositorios". Claro, isso é preciso para que os métodos do repositório se tornem visíveis no contexto da classe que está utilizando o recurso.

Após criarmos o método que irá até o banco de dados e fará a verificação da existência ou não do usuário, é chegado o momento de implementarmos o comportamento de autenticação no *controller*. Lembre-se, estes "caras" são os responsáveis por gerir todas as requisições que chegam ao *framework*.

Continuando, navegue até o diretório *Controllers* e, lá, adicione um novo *controller*. Note que, na janela exibida como o resultado desta ação, o Visual Studio já traz consigo o sufixo *Controller*. Como nomearemos nosso controlador como "Usuarios", teremos o seguinte nome exibido no campo *Controller name*: **UsuariosController**. Na mesma janela, no campo *Template*, selecione a opção "*Empty MVC controller*". Esta é mais uma das CoC's das quais falamos no segundo capítulo desta obra. O

ASP.NET MVC utiliza este sufixo para identificar de forma automática aquela classe como controladora e automaticamente herdar os comportamentos da classe *Controller* da BCL (Base Class Library). A figura 7.1 apresenta esta parametrização.

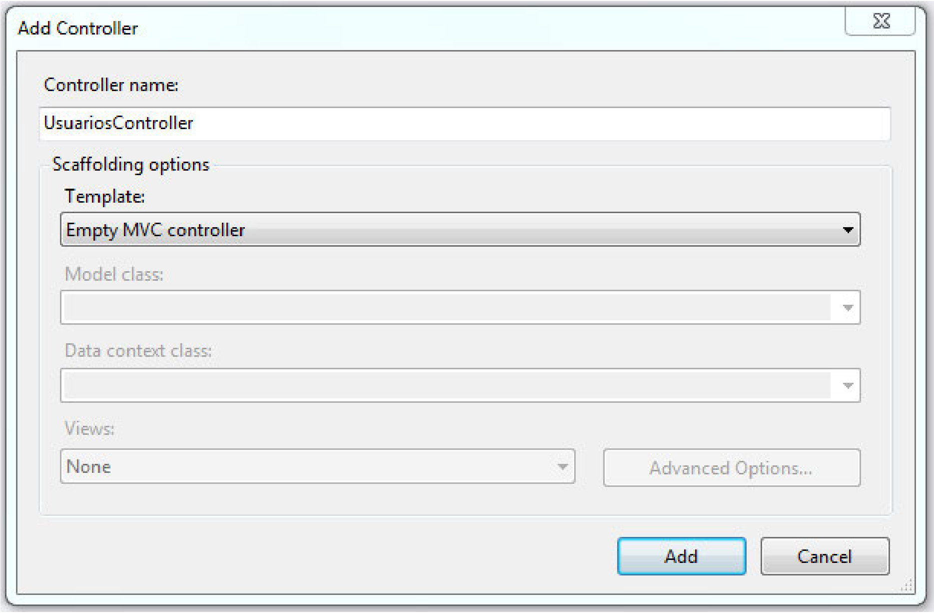


Figura 7.1: Adicionando o controller de usuários

Substitua agora a *action* "Index" do *controller* recém-criado pela "Autenticacao-DeUsuario", apresentada pela listagem 2.

Listagem 7.2 - Criando o método AutenticacaoDeUsuario:

```
[HttpGet]
public JsonResult AutenticacaoDeUsuario(string Login, string Senha)
{
    if (RepositorioUsuarios.AutenticarUsuario(Login, Senha))
    {
        return Json(new {
            OK = true,
            Mensagem = "Usuário autenticado. Redirecionando..." },
            JsonRequestBehavior.AllowGet);
    }
```

```
    else
    {
        return Json(new {
            OK = false,
            Mensagem = "Usuário não encontrando. Tente novamente." },
            JsonRequestBehavior.AllowGet);
    }
}
```

Principais aspectos do código apresentado pela listagem 2:

- Na primeira linha decoramos a *action* "AutenticacaoDeUsuario" com o atributo `[HttpGet]`, indicando que o método apenas devolverá dados para o chamador. Se não decorássemos a *action* com este atributo (ou outro qualquer), o ASP.NET MVC automaticamente imprimirá o comportamento de `[HttpGet]`;
- Ao receber os dados de entrada, utilizamos o método "AutenticarUsuario" do "RepositorioUsuarios", criado na listagem 1;
- Ao receber a resposta da verificação (veja o *if*), retornamos em formato JSON o *status* do processo: sucesso (repare na linha `return Json(new { OK = true, Mensagem = "Usuário autenticado. Redirecionando..."}, JsonRequestBehavior.AllowGet);`) ou falha (note a linha `return Json(new { OK = false, Mensagem = "Usuário não encontrando. Tente novamente."}, JsonRequestBehavior.AllowGet);`). Bacana, não?! Simples e direto.

A pergunta que pode ter se formado em sua cabeça neste instante é: a resposta volta para quem? Veja, no caso da aplicação "Cadê meu médico?", utilizaremos uma chamada assíncrona com jQuery para disparar o processo de verificação da existência ou não do usuário, portanto, quem receberá a resposta final do *controller* será o objeto jQuery que originou o processo. A partir daí, será simples exibir os dados para o usuário final. Pois bem, já que possuímos a infraestrutura de *backend* pronta, precisamos montar a estrutura de *frontend*.

Prosseguindo com o processo de criação de nossa área protegida, abra o arquivo "HomeController" e em seu interior, adicione uma nova *action* que retorna um *ActionResult* e, na sequência, nomeie-a como "Login". A listagem 3 apresenta a

estrutura da mesma.

Listagem 7.3 - Action que retornará a view de login:

```
public ActionResult Login()
{
    ViewBag.Title = "Seja bem vindo(a)";
    return View();
}
```

Clicando com o botão direito sobre o método de retorno (`View()`) e selecionando a opção "*Add view...*", você será apresentado à já conhecida janela de parametrização de *views*. Desta vez, não adicionaremos qualquer configuração a ela, tão pouco herdaremos algum padrão. O que faremos nesta janela apenas, será desmarcar a opção "*Use a layout or master page*" — sim, criaremos uma *view* fora do padrão. O código desta nova *view* pode ser visualizado através da listagem 4.

Listagem 7.4 - Tela de login do sistema:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Cade meu medico? - @ViewBag.Title</title>
    <link href="@Url.Content("~/Content/bootstrap/bootstrap.min.css")"
        rel="stylesheet" />
    <style>
        body {
            padding-top: 60px;
        }
    </style>
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <a class="navbar-brand" href="#">
                    Cade meu medico? - Área Administrativa
                </a>
            </div>
            <div class="collapse navbar-collapse">
```

```
                <ul class="nav navbar-nav">
                    <li id="menuMedicos">

                    </li>
                </ul>
            </div>
        </div>
    </div>

    <div class="container">
        <div class="jumbotron">
            <div class="container">
            <h1>Quem é você?</h1>
            <p> </p>
            <div id="status"></div>
            <p> </p>
            <div>
                <form id="form-login" name="form-login"
                    method="get"
                    action="javascript:function()">

                    <label for="txtLogin">Login</label> 
                    <input type="text" id="txtLogin" name="txtLogin" />
                     
                    <label for="txtSenha">Senha</label> 
                    <input type="password" id="txtSenha"
                            name="txtSenha" /> 
                    <a id="botao-entrar" class="btn btn-primary btn-lg">
                        Entrar
                    </a>

                </form>
            </div>
            </div>
        </div>
        <p style="text-align:center;">
            @Html.ActionLink(
                "Esqueceu sua senha?",
                "EsqueceuSuaSenha",
                "Home",
                null,
```

```
                new { @style = "text-align: center;" })
        </p>
    </div>

    <script src="@Url.Content("~/Scripts/jquery-2.0.2.min.js")"></script>
    <script src="@Url.Content("~/Scripts/bootstrap.min.js")"></script>
</body>
</html>
```

O resultado da renderização do código da *view Razor* apresentado pela listagem 4 pode ser visualizado através da figura 7.2.



Figura 7.2: View de login do sistema Cadê meu médico?

Agora que possuímos parte de nossa infraestrutura de *frontend* necessária pronta, podemos seguir com a segunda parte deste quebra-cabeças, isto é, criar os comportamentos. Para isso, criaremos um novo arquivo Javascript no diretório "Scripts" de nossa aplicação. O processo de criação do arquivo segue o mesmo padrão já utilizado para criar outros tipos de arquivos, ou seja, botão direito no diretório `Scripts > Add > New Item`. Selecione à esquerda a opção "*Web*" e à direita, "*JavaScript File*". Nomearemos sugestivamente este arquivo como "Login.Home.js". Em seu interior, adicione o código apresentado pela listagem 5.

Listagem 7.5 - Código Javascript que dispara o processo de validação:

```
$(document).ready(function () {
    $("#status").hide();
    $("#botao-entrar").click(function () {
```

```
        $.ajax({
            url: "/Usuarios/AutenticacaoDeUsuario",
            data: { Login: $("#txtLogin").val(),
                    Senha: $("#txtSenha").val() },
            dataType: "json",
            type: "GET",
            async: true,
            beforeSend: function () {
                $("#status").html("Estamos autenticando o usuário.
                                   Só um instante...");
                $("#status").show();
            },
            success: function (dados) {
                if (dados.OK) {
                    $("#status").html(dados.Mensagem)
                    setTimeout(function () { window.location.href =
                                            "/Home/Index" }, 5000);
                    $("#status").show();
                }
                else {
                    $("#status").html(dados.Mensagem);
                    $("#status").show();
                }
            },
            error: function () {
                $("#status").html(dados.Mensagem);
                $("#status").show()
            }
        });
    });
});
```

O que fazemos no código apresentado pela listagem 5 é:

1) Realizamos uma chamada assíncrona utilizando o método jQuery Ajax;

2) Informamos o *controller* e a *action* que deverão ser executadas de forma assíncrona, o tipo de retorno e o tipo de operação;

3) Informamos ao navegador o que deverá ser realizado antes da chamada ser realizada (`beforeSend`), caso a resposta seja de sucesso (`success`) ou falha (`error`). Simples!

Para testar o funcionamento de tudo que foi feito até aqui, precisamos adicionar um usuário qualquer no banco de dados da aplicação para que possamos simular a situação de sucesso (usuário encontrado), certo?! Faremos tal procedimento adicionando o registro de usuário no banco de dados de forma direta com a cláusula `INSERT`, apresentada pela listagem 6. Se você utilizou o script de criação do banco de dados disponibilizado pode pular esse passo, o script adiciona esse registro automaticamente na criação do banco de dados.

Listagem 7.6 -

:

```
Insert Into Usuarios
    (Nome, Login, Senha, Email)
Values
    ('Administrador', 'admin',
    '40BD001563085FC35165329EA1FF5C5ECBDBBEEF', 'admin@cdmm.com')
```

Todos os mecanismos estão prontos e funcionando e o usuário de teste devidamente cadastrado. Estamos prontos para realizar os testes. No primeiro teste, informaremos usuário e senha incorretos e, então, verificaremos o comportamento do sistema. A figura 7.3 apresenta a resposta do sistema.

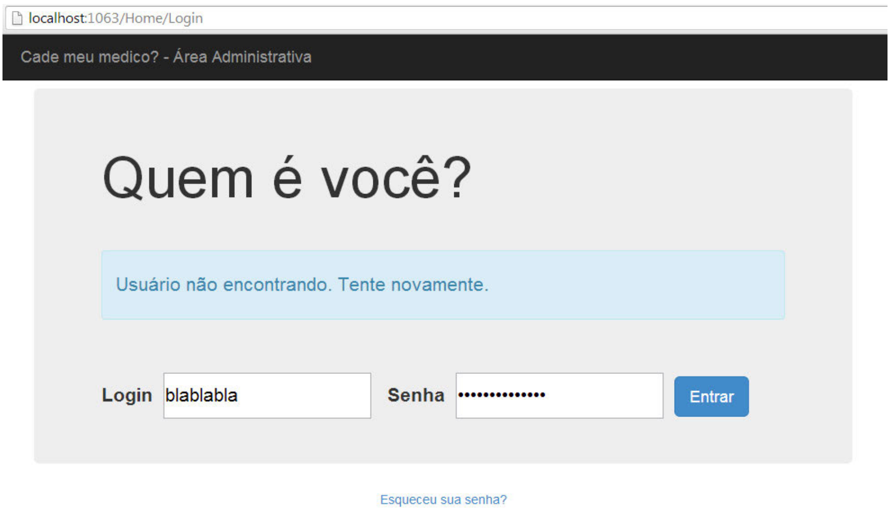


Figura 7.3: Resposta do sistema a entradas incorretas para o sistema

Quando adicionamos as informações corretas — para que você possa testar, é importante saber que a senha é "123" —, compatíveis com as que se encontram cadastradas no banco de dados, obtemos a resposta apresentada pela figura 7.4.

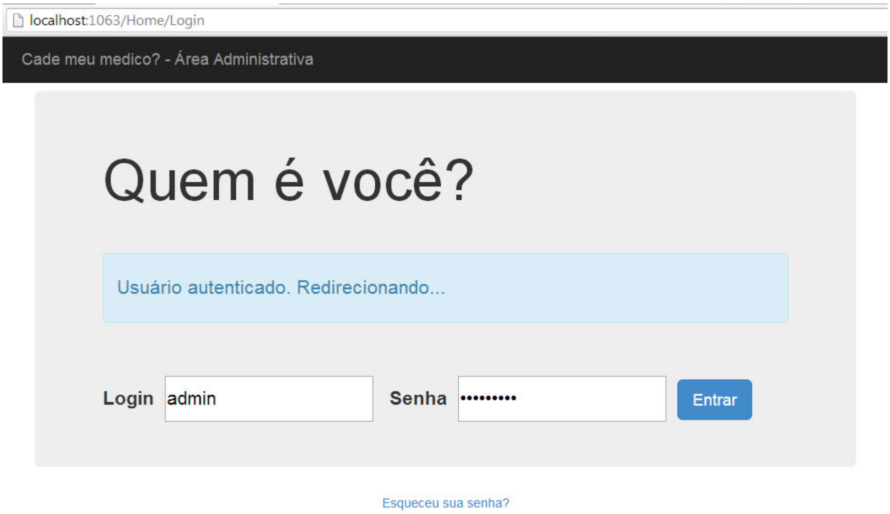


Figura 7.4: Resposta do sistema para uma entrada correta

O teste final a ser realizado para que possamos verificar se de fato tudo está funcionando conforme o esperado é a verificação da existência do *cookie* que armazena a informação do usuário recém-autenticado. A figura 7.5 comprova que tudo está correto.

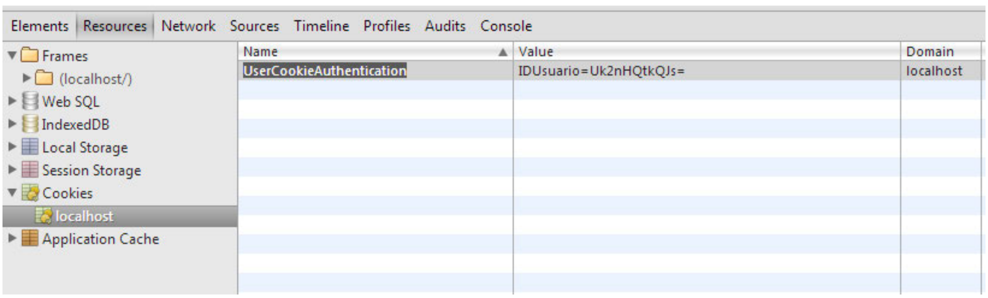


Figura 7.5: Verificando se o cookie de apoio foi criado corretamente

Ok, mecanismo de login implementando.

7.3 Apesar do login, nada é bloqueado. E agora?!

Muito embora tenhamos "cercado" nossa aplicação através da implementação do mecanismo de autenticação via *login*, facilmente podemos notar uma falha de segurança grave: se invocarmos determinado recurso de forma direta (pela URL, por exemplo), mesmo não estando autenticado no sistema, o recurso se mostrará disponível. Este é o caso, por exemplo, do *controller*/*action* "Home/Index", que deveria ser exibido apenas para usuários registrados.

Neste contexto, um recurso do ASP.NET MVC disponível desde a versão 2 do *framework* se apresenta como uma boa alternativa para solucionar o problema citado no parágrafo anterior: os *action filters* (em português brasileiro, filtros de ação).

7.4 Filtros de ação

Filtros de ação são, de forma direta, comportamentos específicos que podem ser atrelados a *actions* e *controllers*. Através de uma forma declarativa simples (entenda-se atributos), é possível incorporar robustez a natureza das aplicações.

Um dos grandes benefícios proporcionados pelos filtros de ação é a melhoria significativa do *design* (arquitetura) da aplicação, uma vez que permite retirar de *actions* e *controllers* comportamentos que não são restritos a eles. Um exemplo claro desta situação encontra-se justamente no que faremos a seguir para a aplicação "Cadê meu médico?". Note, validar a permissão ou não de determinado usuário para acessar a página inicial da área administrativa (**Home/Index**) não é função da *action* "Index", certo? Desta forma, a solução é encapsular tal comportamento em um filtro. Vamos à sua construção?

7.5 Implementando o filtro 'AutorizacaoDeAcesso'

O comportamento que pretendemos imprimir para nossos *controllers* e *actions* é: se determinado usuário tentar acessar um recurso de forma direta, o sistema deverá repeli-lo, ou seja, redirecionar o usuário para a tela de autenticação do sistema com o caminho pretendido preservado na barra de navegação.

Criando a classe 'AutorizacaoDeAcesso'

Como primeira medida no sentido estruturarmos nosso filtro de ação, adicionaremos um diretório chamado "Filtros" à árvore de diretórios de nosso projeto. Em seu interior, adicionaremos uma nova classe, chamada "AutorizacaoDeAcesso". Se tudo correu bem neste processo, você deverá estar visualizando uma tela semelhante à apresentada pela figura 7.6.

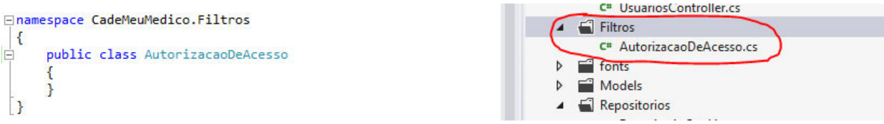


Figura 7.6: Estrutura gerada para o filtro AutorizacaoDeAcesso

Agora, deixaremos o arquivo recém-criado temporariamente de lado. Faremos isso porque precisamos implementar dois métodos auxiliares, ambos no repositório de usuários ("RepositórioUsuarios") que criamos na segunda seção deste capítulo. Estes métodos nos ajudarão a verificar se o usuário já se encontra autenticado no sistema e, com base nesta resposta, decidirmos se rejeitamos ou permitimos o acesso.

Os métodos aos quais nos referimos são apresentados na sequência ao longo da listagem 7.

Listagem 7.7 - Métodos de apoio para verificação do status do usuário:

```
public static Usuario RecuperaUsuarioPorID(long IDUsuario)
{
    try
    {
        using (EntidadesCadeMeuMedicoBD db =
            new EntidadesCadeMeuMedicoBD())
        {
            var Usuario =
                db.
                Usuarios.
                Where(u => u.IDUsuario == IDUsuario).
                SingleOrDefault();
            return Usuario;
        }
    }
```

```
    catch (Exception)
    {
        return null;
    }
}

public static Usuario VerificaSeOUsuarioEstaLogado()
{
    var Usuario = HttpContext.
            Current.
            Request.
            Cookies["UserCookieAuthentication"];
    if (Usuario == null)
    {
        return null;
    }
    else
    {
        long IDUsuario = Convert.
        ToInt64(RepositorioCriptografia.
        Descriptografar(Usuario.Values["IDUsuario"]));

        var UsuarioRetornado = RecuperaUsuarioPorID(IDUsuario);
        return UsuarioRetornado;

    }
}
```

Os nomes de ambos os métodos apresentados pela listagem 7 já dão ideia de suas funções no contexto da aplicação. O primeiro tenta recuperar o registro único de usuário baseado no código proveniente do parâmetro do mesmo. O segundo toma como base o usuário recuperado pelo primeiro método e verifica se o usuário já se encontra autenticado no sistema. Simples e funcional.

Agora possuímos os métodos auxiliares de nosso filtro, portanto, podemos partir para sua implementação de fato. Para isso, volte ao arquivo do filtro que criamos alguns parágrafos atrás e substitua seu conteúdo por aquele apresentado pela listagem 8.

Listagem 7.8 - Implementando o filtro de ação:

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using CadeMeuMedico.Repositorios;

namespace CadeMeuMedico.Filtros
{
    [HandleError]
    [AttributeUsage(AttributeTargets.Class |
                    AttributeTargets.Method,
                    Inherited = true,
                    AllowMultiple = true)]
    public class AutorizacaoDeAcesso :
                 ActionFilterAttribute
    {
        public override void OnActionExecuting(
            ActionExecutingContext FiltroDeContexto)
        {
            var Controller = FiltroDeContexto.
                             ActionDescriptor.
                             ControllerDescriptor.
                             ControllerName;
            var Action = FiltroDeContexto.
                         ActionDescriptor.
                         ActionName;

            if (Controller != "Home" || Action != "Login")
            {
                if (RepositorioUsuarios.
                    VerificaSeOUsuarioEstaLogado() == null)
                {
                    FiltroDeContexto.
                        RequestContext.
                        HttpContext.
                        Response.
                        Redirect("/Home/Login?Url=" +
                            FiltroDeContexto.H
                            ttpContext.
                            Request.
                            Url.
```

```
                            LocalPath);
                }
            }
        }
    }
}
```

Nossa primeira providência ao criar o método é decorar a classe como um gerenciador de erros automatizado (vide `[HandleError]`). Na sequência, informamos através da diretiva "AttributeUsage" e seus atributos, qual o comportamento que esperamos encontrar ao utilizar o filtro. Após isso, sobrescrevemos o método "OnActionExecuting" e personalizamos o comportamento da requisição que chega ao *controller*.

Como você pode notar, através de uma verificação simples (*if*), verificamos se a requisição é por um recurso protegido por usuário e senha. Se não for, o acesso ao recurso é liberado, caso contrário, direcionamos o usuário para a área de entrada (*login*) preservando o endereço buscado pelo usuário na requisição original.

Criando o controller base

Agora, para que o processo possa ser completado, precisamos imprimir o mesmo comportamento para todos os *controllers* da aplicação. Como faremos isso? Simples: iremos decorar o *controller* base com o filtro recém-criado e, a partir daí, para cada novo *controller*, herdaremos de *controller* base.

Para isso, adicionaremos um novo *controller* chamado "Base" em seu respectivo diretório e, em seu interior, implantaremos o código apresentado pela listagem 9.

Listagem 7.9 - Comportamento do controller base:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using CadeMeuMedico.Filtros;

namespace CadeMeuMedico.Controllers
{
    [AutorizacaoDeAcesso]
```

```
    public class BaseController : Controller
    {
        protected override void OnActionExecuting(
            ActionExecutingContext filterContext)
        {
            base.OnActionExecuting(filterContext);
        }

    }
}
```

Basta agora, nos *controllers* da aplicação, mudar a herança. Ao invés de herdar diretamente de *Controller*, estes controladores deverão herdar de *BaseController*, conforme apresenta o exemplo da listagem 10.

Listagem 7.10 - Modificando a herança dos controladores:

```
public class HomeController : BaseController
    {
        //
        // GET: /Home/

        public ActionResult Login()
        {
            ViewBag.Title = "Seja bem vindo(a)";
            return View();
        }

        public ActionResult Index()
        {
            return View();
        }

    }
```

Agora podemos testar o comportamento de nossa aplicação. Para isso, remova os *cookies* de seu navegador e, sem realizar *login*, tente acessar o endereço " http://localhost: `[sua_porta]/Home/Index`". Se tudo correu bem, você será redirecionado para a tela de *login* com o endereço buscado preservado na barra de navegação, conforme apresenta a figura 7.7.

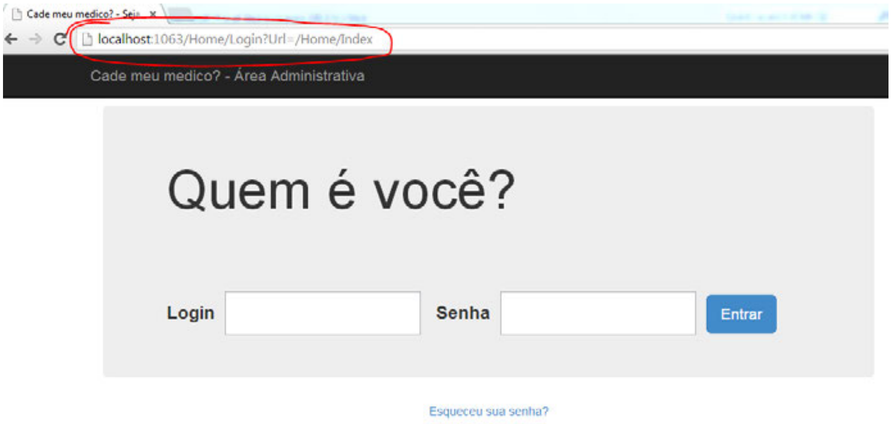


Figura 7.7: O mecanismo de validação em funcionamento

7.6 Concluindo e desafiando

Com este conglomerado de conceitos, nosso objetivo é mostrar uma das formas possíveis para que se possa proteger as aplicações ASP.NET MVC de acessos indevidos. É claro que estes conceitos e recursos não param por aqui. Existe uma ampla gama de recursos para se trabalhar com estes conceitos disponíveis no *framework* MVC. Você pode encontrar um texto apresentando estes recursos através deste link: http://bit.ly/segurancamvc.

Desafio: imagine agora que você precise adicionar novos conteúdos que não necessariamente são protegidos por senha (como o recurso de "Recuperar Senha", por exemplo). Onde você alteraria a estrutura da aplicação para que a aplicação não recusasse o acesso ao mesmo?! Mãos à obra.

Capítulo 8

Publicando sua aplicação

Se você seguiu à risca todos os exemplos apresentados por este livro, neste momento deve possuir em mãos boa parte da aplicação "Cadê meu médico?". Se você acessar o endereço [*lugar onde disponibilizaremos o exemplo completo*], você poderá encontrar a aplicação completa para seguir com o conteúdo deste capítulo — **Publicação da aplicação**.

Optamos por deixar a apresentação do processo de publicação da aplicação como último capítulo não pelo fato de ser menos importante que os demais mas sim, por uma questão lógica muito simples: como podemos publicar algo sem que este algo exista? O processo de publicação de aplicações é importante, principalmente nos dias atuais em que tantos tipos de ambientes distintos se apresentam como opções para *host*.

Neste capítulo apresentaremos alguns cenários e ferramentas disponíveis no mundo da tecnologia para que possamos publicar aplicações ASP.NET, especialmente, ASP.NET MVC 4. Especialmente neste capítulo, manteremos o foco na publicação para um ambiente específico, uma plataforma específica de computação em

nuvem, criada e mantida pela Microsoft — o Windows Azure.

8.1 Pré-requisitos para hospedar aplicações ASP.NET MVC 4

Para que as aplicações ASP.NET MVC 4 possam funcionar de forma adequada em um ambiente remoto, é preciso que haja neste ambiente um *set* de configurações já realizadas e certo grupo de ferramentas disponíveis. A lista a seguir apresenta o conjunto mínimo de requisitos que devem estar disponíveis no servidor de aplicação para que tudo funcione perfeitamente.

- **Windows (7, 7 com SP1, Server 2003 com SP2, Server 2008, Server 2008 R2, Server 2008 R2 com SP1, Vista com SP1, XP com SP3)**: você já conhece o Windows, certo?
- **.NET Framework 4(+)**: o .NET Framework é o modelo de programação completo e consistente da Microsoft para a criação de aplicativos que oferecem uma experiência visualmente surpreendente aos usuários, comunicação segura e sem interferências e a capacidade de modelar uma variedade de processos de negócios;
- **ASP.NET MVC 4**: *framework* ASP.NET MVC em sua versão 4;
- **Internet Information Services 6.0(+)**: o *Internet Information Services* (ou simplesmente IIS) é o componente que faz do servidor um servidor de aplicações web;
- **Microsoft Data Access Components 2.8(+)**: um conjunto de DLL's utilizada pelo ADO para realizar o acesso a dados;
- **Windows Installer 3.1(+)**: é um componente que automatiza o processo de instalação dos recursos no sistema operacional;
- **Internet Explorer 5.01(+)**: é o navegador web da Microsoft. Este componente vem nativamente instalado no Windows;
- **Pentium 1 GHz(+)**: *clock* de processador mínimo esperado;
- **RAM 512 MB(+)**: quantidade mínima de memória RAM exigida;

- **x86, x64 e ia64**: arquiteturas suportadas pelos recursos anteriores.

Você pode realizar o *setup* do ambiente manualmente (instalando manualmente cada um dos componentes mencionados na lista anterior) ou pode utilizar uma ferramenta disponibilizada gratuitamente pela Microsoft, o WebPI (acrônimo para *Web Platform Installer*), que automatiza todo o processo de instalação e configuração de ambiente no servidor remoto, gerenciando inclusive, as dependências de cada elemento. Você pode encontrar mais informações sobre esta ferramenta seguindo este link: http://bit.ly/webpiaspnetmvc.

8.2 Ambientes de host

O mercado de tecnologia disponibiliza uma série de ambientes distintos para que seja possível hospedar aplicações ASP.NET (MVC). São os principais:

- **Hospedagem compartilhada (a mais comum)**: existem praticamente infinitas opções no Brasil ou fora dele para realizar o *host* de aplicações ASP.NET. Uma rápida busca no Google dará veracidade a esta afirmação. O modelo tradicional de *host* (conhecido como "compartilhado") contempla múltiplos websites em um mesmo ambiente, compartilhando todos os recursos de *hardware* e *software* disponíveis no servidor;
- **Virtual Private Server (VPS)**: neste caso, o *deploy* da aplicação ASP.NET é realizado em um servidor virtual dedicado para a aplicação. Nestes ambientes, de forma geral, a configuração fica por conta de quem contrata;
- **Servidor dedicado**: neste caso, temos um servidor físico. Este servidor terá todos os recursos disponíveis para executar o servidor de aplicação e "rodar" a aplicação ASP.NET MVC. A única diferença em relação às VPS's é o fato de que um é virtualizado e o outro é físico;
- **Ambientes de Cloud**: nos ambientes de computação em nuvem (em ampla expansão no mercado) existem diferentes modelos para *host* de aplicações ASP.NET MVC. Discutiremos alguns deles mais adiante neste capítulo.

Para cada um destes cenários, para que as aplicações possam ser enviadas ao servidor remoto, existem diferentes abordagens, tais como FTP, SFTP, Web Deploy etc. Para cada um destes modelos, existem diversas ferramentas para realizar este envio.

Para FTP, por exemplo, podemos citar o próprio Visual Studio Express, FileZilla, CuteFTP, dentre outros. Se a escolha for pelo Visual Studio, existe ainda a opção de gerar um pacote de publicação local (para publicar mais tarde) utilizando outra ferramenta qualquer.

Para demonstrar a publicação da aplicação "Cadê meu médico?", utilizaremos o próprio Visual Studio. Você entenderá o porquê no momento adequado.

8.3 Computação em nuvem. Por quê?

O conceito de computação em nuvem, indiscutivelmente, revolucionou o mercado de tecnologia. A transformação de ambientes e recursos complexos (que em momento anterior, eram privilégios apenas de grandes empresas) em serviços com baixo custo e portanto, acessíveis a qualquer tipo de empresa, foi o grande "pulo do gato" no mundo da tecnologia.

A ideia aqui é que, assim como arquivos, servidores e demais serviços associados fiquem distribuídos ao redor do mundo e que, de uma forma "mágica", proporcionem alta disponibilidade, tolerância falhas, conexões resilientes, segurança, performance de aplicações e principalmente, baixo custo — também em função da diminuição da complexidade para gerir estes serviços e recursos.

Para facilitar o entendimento dos conceitos relacionados à computação em nuvem, os serviços e recursos oferecidos por ela são subdivididos em três grandes verticais: *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) e *Software as a Service* (SaaS).

No primeiro grupo (**IaaS**) encontram-se agrupados todos os serviços e recursos associados diretamente ao modelo de infraestrutura, como por exemplo, servidores virtuais. É claro que, neste primeiro grupo, o esforço impresso por quem está adquirindo o serviço de nuvem é maior, uma vez toda a configuração do ambiente deve ser realizada por ele. De forma geral, buscam serviços nesta vertical empresas que desejam possuir um alto grau de personalização de ambiente.

O segundo grupo (**PaaS**) incorpora serviços e recursos de nível mais alto. Aqui, o cliente contrata uma plataforma já pronta, pré-configurada, e simplesmente envia a aplicação para que execute neste ambiente. Todo esforço de manutenção do ambiente fica por conta do fornecedor da estrutura de computação em nuvem. Fica sob a responsabilidade do cliente apenas realizar o *deploy* da aplicação. De forma geral, quem busca estes ambientes são empresas de *software*, que não querem dispender esforços com a parte de infraestrutura.

O terceiro grupo (**SaaS**) nada mais é do que o resultado final, após a publicação. São os *softwares* construídos especificamente para a plataforma de nuvem e que tem comportamento ótimo neste ambiente. Exemplos: Office 365, Google Docs, Educa-Net etc. "Cadê Meu Médico?" também será um *software* como serviço no fim deste capítulo.

A figura 8.1 apresenta a relação entre as vertentes de computação em nuvem e o esforço que deverá ser dispendido entre quem compra e quem fornece estes serviços.

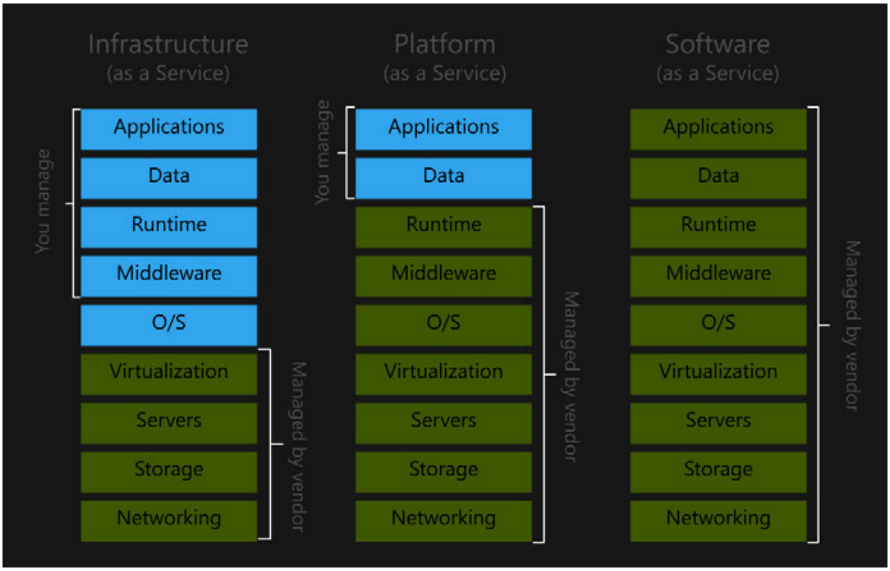


Figura 8.1: Pilha de responsabilidades da computação em nuvem

8.4 O Windows Azure

O Windows Azure é uma plataforma de computação em nuvem, criada e mantida pela Microsoft. No mercado de computação em nuvem, a Microsoft é lider, dividindo a honraria com a AWS (Amazon Web Services).

A palavra "plataforma" não é utilizada aqui por acaso. Windows Azure oferece uma ampla gama de recursos que atendem à grande maioria das demandas de mercado com qualidade e bom preço, seja para o desenvolvimento de aplicações ou para a área de infraestrutura. Para fins de testes, a Microsoft oferece como degustação, um *voucher* no valor de US$ 200,00 durante três meses. Você poderá encontrar todas

as informações relacionadas ao Windows Azure através do site oficial do produto: http://windowsazure.com/.

8.5 Windows Azure Websites

Um dos grandes recursos disponíveis no Windows Azure é o que a Microsoft chama de "Windows Azure Websites".

Em linhas gerais, Windows Azure Web Sites é um novo ambiente disponibilizado dentro da plataforma Azure com características específicas para executar aplicações web de pequeno e médio portes. Ao criar um novo website neste modelo, assim como em uma hospedagem compartilhada tradicional, sua aplicação compartilhará recursos (disco, RAM, processamento etc.) com outras aplicações em um mesmo servidor — de alta capacidade, é importante que se diga. A ideia é que, através do WAWS, seja possível realizar o deploy de aplicações web de forma simplificada e rápida, utilizando o controlador de código de sua preferência. Alguns aspectos importantes a serem considerados em relação ao WAWS são:

- **Git ou TFS deployment**: com WAWS é possível utilizar Git ou TFS para efetuar a publicação de suas aplicações de forma direta. Com algumas poucos comandos e cliques, é possível habilitar o ambiente para deploy através destas ferramentas;

- **Escolha da publicação ativa**: o portal do WAWS armazena cronologicamente às últimas publicações realizadas. Com alguns cliques você pode selecionar qual dentre elas é aquela que deve estar ativa;

- **Gerenciamento de configuração de aplicações**: gerenciar os diversos valores e atributos de arquivos de configurações (web.config's) é sempre uma dor de cabeça à parte. Com WAWS este problema foi reduzido drasticamente através de um mecanismo simples de parametrização;

- **Log's do site**: WAWS armazena log's dos sites lá armazenados. Você pode monitorar a "saúde" de sua aplicação analisando-os. Este acesso pode ser realizado via client de FTP ou via ferramenta de linha de comando;

- **Bases de dados MySQL**: WAWS suporta também bancos de dados MySQL. Você criar uma nova base de dados ou, ainda, utilizar uma base já existente

para conectar sua aplicação. Para que MySQL pudesse funcionar com performance e segurança no modelo de cloud, a Microsoft estabeleceu uma parceria com uma empresa especializada, chamada ClearDB;

- **Escalabilidade**: Windows Azure Web Sites disponibiliza dois modelos possíveis para conseguir escala: compartilhado e reservado. No modo compartilhado é possível aumentar o número de processos referentes à aplicação web, enquanto no reservado é possível passar a aplicação para máquinas virtuais dedicadas e subir o número de máquinas que respondem para as aplicações lá hospedadas;

- **Segurança**: WAWS trabalha em modo *full trust*. Isto significa dizer que todas as aplicações que "rodam" em ambiente compartilhado encontram-se completamente isoladas umas das outras. Este é um critério importante a ser avaliado no momento de planejar seu website;

- **Linguagens e frameworks**: você já deve ter ouvido falar que WAWS suporta nativamente ASP.NET, ASP.NET MVC, PHP, Node.js. Talvez o que você não saiba é que suas aplicações ASP clássicas e 2.0 também são suportadas, logo, seus códigos legados são bem-vindos dentro do Windows Azure Web Sites;

- **Web Deploy**: outra característica importante agregada ao WAWS é a aceitação de publicação de aplicações via *web deploy*. Assim, se você utiliza as ferramentas da Microsoft para gerenciamento de código (VS 2010, VS 2012, VS 2013 ou WebMatrix), poderá publicar diretamente através delas no Windows Azure;

- **NuGet**: ao realizar o deploy de sua aplicação no WAWS, não é mais necessário enviar os pacotes de forma agregada. O Windows Azure se encarrega de realizar tal tarefa em tempo de execução no momento da implantação; **Domínios personalizados**: uma dificuldade encontrada por boa parte dos desenvolvedores web que utilizam web roles no Windows Azure é a criação de domínios personalizados. Com Windows Azure Web Sites, a especificação de domínios personalizados tornou-se extremamente simples, podendo utilizar o próprio portal de administração para este fim. Entretando, é importante observar que, para que esta feature funcione, o web site deve estar "rodando" em modo reservado;

- **Certificados**: Windows Azure Web Sites oferecem suporte nativo para utilização de SSL para domínios externos e internos;

- **IIS8, Windows Server 2012 e .NET 4.5**: para este primeiro preview, WAWS não implementa IIS8, Windows Server 2012 e .NET 4.5. Muito embora as novas versões ainda não sejam suportadas, estas deverão ser automaticamente implementadas à medida que se tornarem estáveis. Hoje o ambiente é: IIS 7.5, Windows Server 2008 R2 e .NET 4.5.

Uma pergunta que deve estar perturbando sua cabeça neste instante deve ser: "Se o Windows Azure já possuía as web roles, o que justifica a existência de mais um modelo pra host de aplicações web?". Acertei?

A resposta é simples: *Web roles* são ambientes que permitem alto grau de personalização através de *startup tasks*. O objetivo das *web roles* é disponibilizar um ambiente para *host* de aplicações web mais robusto e totalmente personalizável, ao passo que o WAWS apresenta outro viés em termos de demanda: a simplicidade em detrimento da personalização.

Se você precisa de um ambiente personalizado como, por exemplo, implementar características específicas no IIS ou, ainda, adicionar novas configurações de ambiente, WAWS não são para você. Neste caso, recomendamos considerar duas hipóteses: *Web Roles* ou o recurso de máquinas virtuais.

8.6 Publicando a aplicação 'Cadê Meu Médico?'

Você deve estar se perguntando a esta altura: por que tantas informações acerca do Windows Azure Web Sites? Simples. Ele será o ambiente para o qual faremos a publicação da aplicação "Cadê Meu Médico?".

Criando o ambiente de publicação no Windows Azure

Iniciaremos o processo de publicação de nossa aplicação pela criação e preparação do ambiente de *host* no WAWS. Para isso, como primeiro passo, se você não possui uma assinatura do Windows Azure, você deverá criar uma. Para fazê-lo, basta acessar o website do Windows Azure (http://windowsazure.com) e clicar no botão "*Free Trial*". Na tela seguinte, clique no botão "*Try it now*". Ao fazê-lo, você será direcionado para a tela de autenticação do Windows Azure. Você precisará de um *Live ID* para acessar o serviço. Preencha o formulário que se apresentará (incluindo o número de cartão de crédito internacional). Não se preocupe, nada será cobrado em seu cartão. A figura 8.2 apresenta a primeira tela com dados a serem preenchidos.

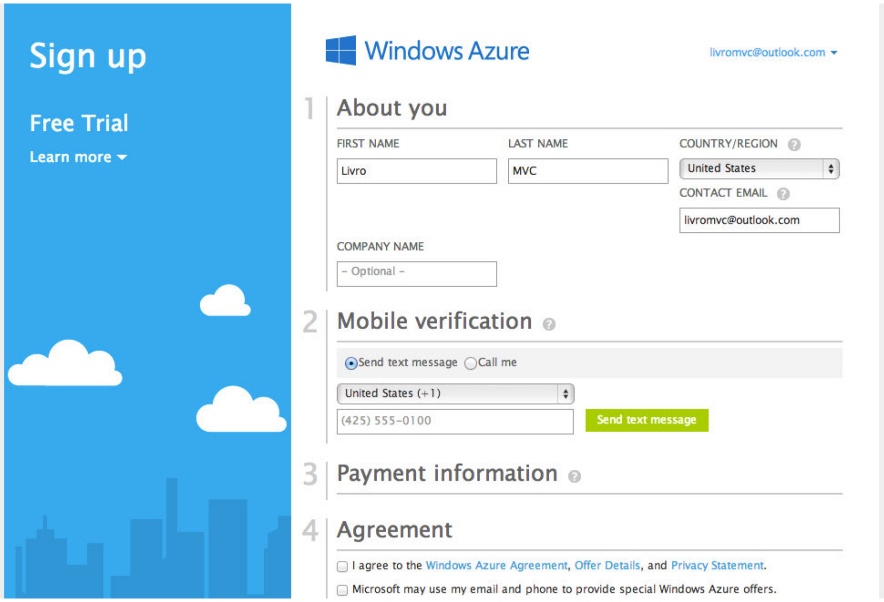


Figura 8.2: Assinando o Windows Azure por 3 meses

Após concluir o processo de assinatura, o Windows Azure liberará uma quota de US$ 200,00 para que você possa utilizar da forma que achar melhor. Claro, utilizaremos para criar nosso ambiente de *host*.

Como próximo passo, acesse o endereço do portal administrativo do Windows Azure: http://manage.windowsazure.com. Lá, informe seus dados o *Live ID* (os mesmos que utilizou para assinar o Windows Azure) e clique em "*Sign In*". Se tudo correu bem, você visualizará uma tela semelhante à apresentada pela figura 8.3.

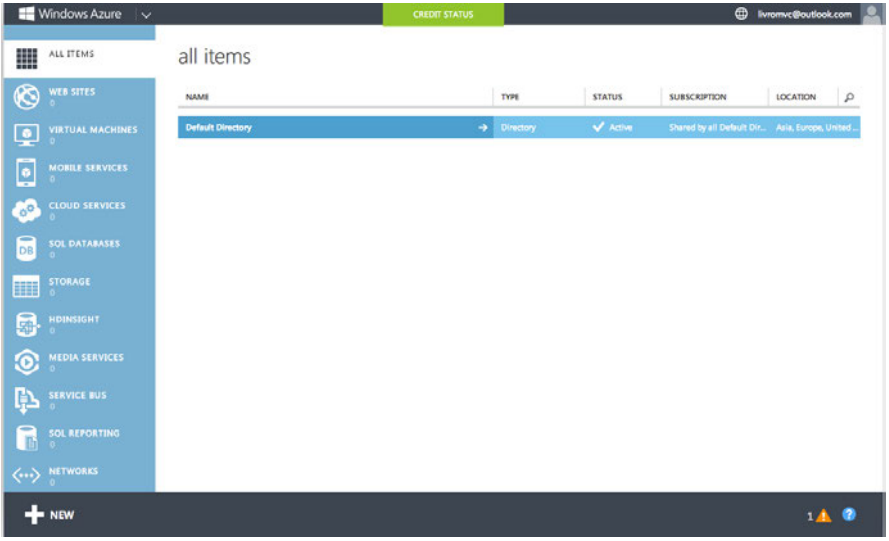


Figura 8.3: Portal de administração do Windows Azure

A figura 8.3 apresenta a tela inicial do portal administrativo do Windows Azure. Note que à esquerda encontra-se o menu principal do portal. Como criaremos um novo site, é claro que clicaremos na opção "*Web sites*". Na tela que se apresentará, você deverá clicar no link "*Create a web site*". Uma janela se abrirá no rodapé com as devidas opções selecionadas. Apenas adicionaremos o nome de nossa "área de *host*" (sugestivamente "livroaspnetmvc"). A região selecionada será "East US". A figura 8.4 apresenta esta tela.

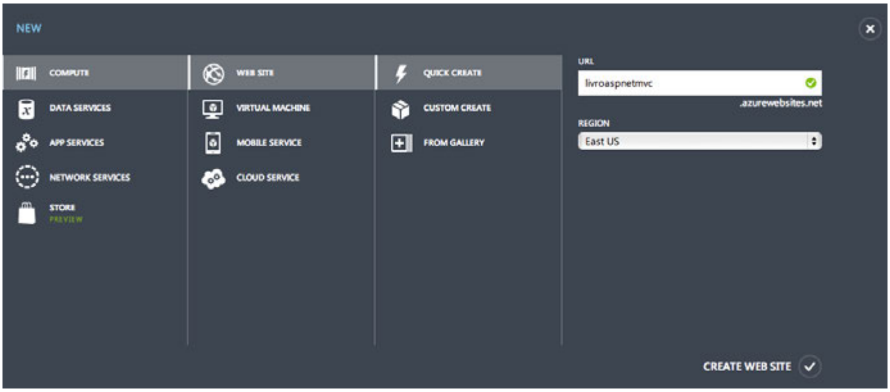


Figura 8.4: Criando um novo web site

Se você reparar bem no rodapé do portal administrativo, verá uma barra de *status* que notifica o usuário final acerca do que está ocorrendo. Se tudo correr bem, você visualizará o novo ambiente criado, conforme apresentado pela figura 8.5.

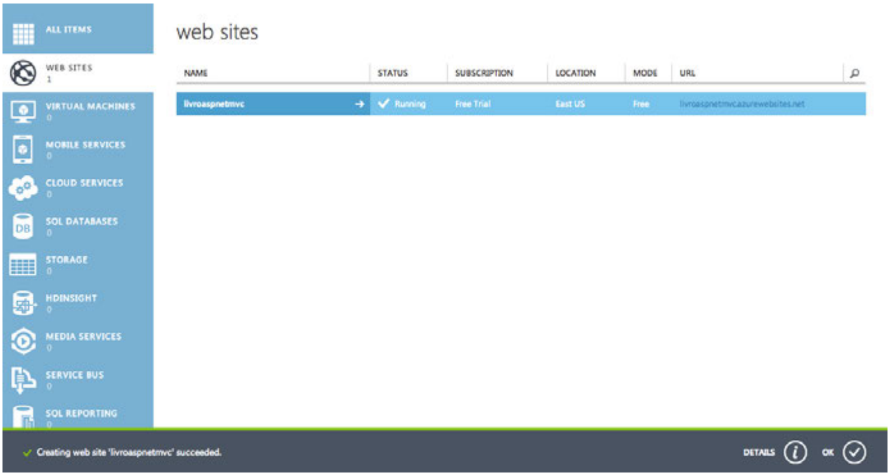


Figura 8.5: Novo ambiente de host criado com sucesso

Pronto! Ambiente de host criado. Para finalizar esta etapa, clique no nome do ambiente de hospedagem. Na tela que se apresenta, clique no link "*Download the publish profile*". Ao fazer isso, um arquivo XML com as configurações de publicação será baixado. Guarde bem o local onde este arquivo será salvo. Ele será utilizado no

processo de publicação. A tela que exibe esta opção pode ser visualizada através da figura 8.7.

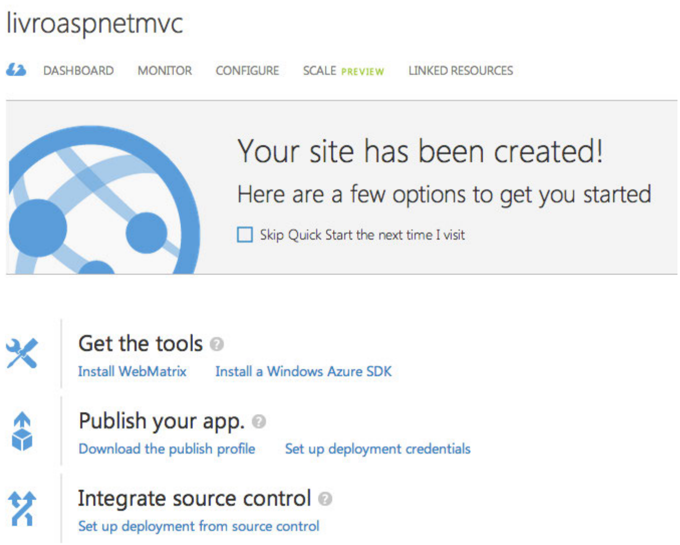


Figura 8.6: Baixando o perfil de publicação

Publicando a aplicação

Agora que já possuímos o ambiente de *host* criado, podemos publicar nossa aplicação. Como você irá constatar, a publicação de uma aplicação no Windows Azure Web Sites com o Visual Studio é extremamente simples. Não poderia ser diferente, já que os produtos são do mesmo fabricante.

Com o projeto "Cadê meu médico?" aberto (mas não em execução), clique com o botão direito sobre o nome do projeto. No menu que se abrirá, selecione a opção "*Publish...*". Ao fazê-lo, uma nova janela de parametrização será exibida. Clique no botão "*Import*", disponível imediatamente a frente do *combo box*. Na janela que se apresentará, selecione a opção "*Import from a publish profile file*" e clique no botão "*Browse*". Navegue até o arquivo de publicação que baixamos quando criamos o ambiente de publicação e clique em "*Open*". Ao selecionar o arquivo e clicar em "*Ok*", automaticamente o Visual Studio alternará a aba de "*Profile*" para "*Connection*", conforme apresenta a figura 8.7.

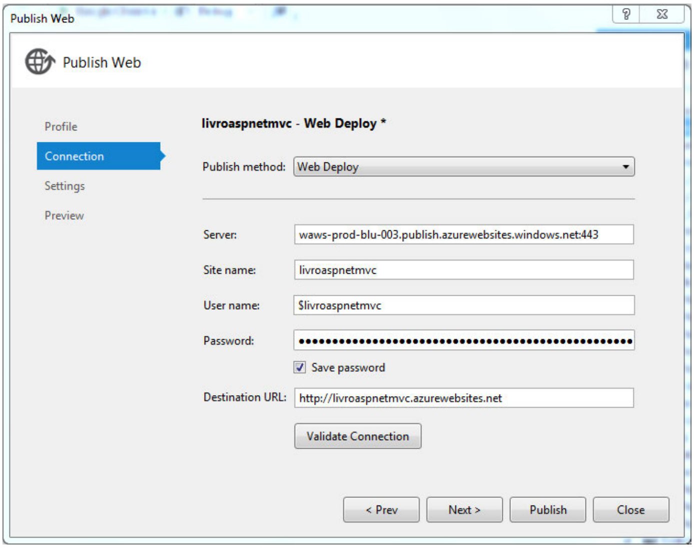


Figura 8.7: Informações de publicação importadas

Ao clicar em "*Next*", o *wizard* alternará para a aba "*Settings*". Aqui poderemos configurar a *string* de conexão para o banco de dados. Neste momento, você poderá indicar um banco de dados remoto ao qual sua aplicação se conectará. Guarde este ponto.

Faremos uma pausa neste momento para publicarmos o banco de dados da aplicação. O que faremos aqui é criar uma instância do banco de dados de "CadeMeuMedicoBD" no *SQL Database* para só depois, voltar ao Visual Studio para apontar a aplicação para este.

Migrando o banco de dados do ambiente local para o online

Voltando ao painel administrativo do Windows Azure, no menu principal do painel, clique na opção "*SQL Databases*". Na tela que se abrir, clique na opção "*Create a SQL Database*". Informe o nome do banco de dados, a edição (selecione "Web"), o tamanho, o *collation* e o servidor onde o banco será criado. A tela parametrizada pode ser visualizada através da figura 8.8.

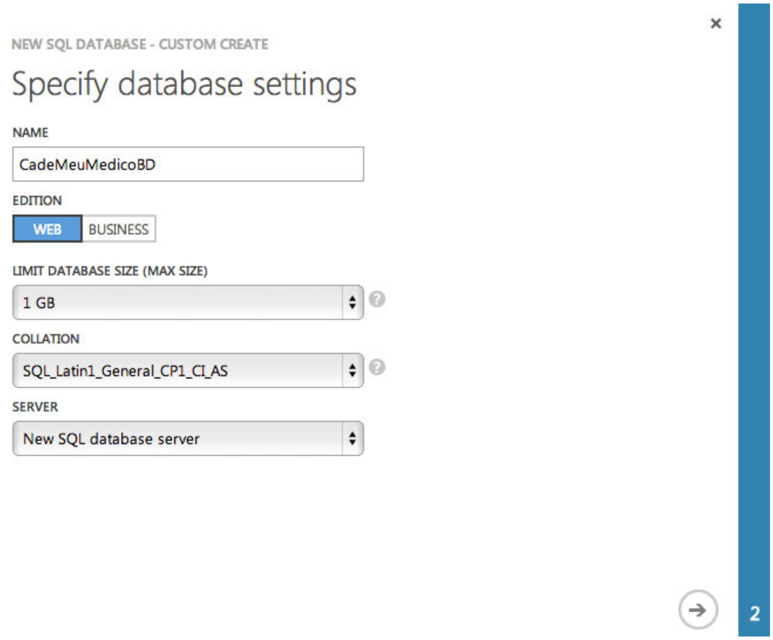


Figura 8.8: Janela parametrizada na criação do banco de dados

Ao clicar na setinha de "Next", uma janela solicitando os dados do servidor de banco de dados será apresentada. Informe usuário e senha de acesso ao servidor, selecione o *datacenter* onde este servidor será alocado e, na sequência, clique em "Ok". A etapa dois parametrizada pode ser visualizada através da figura 8.9.

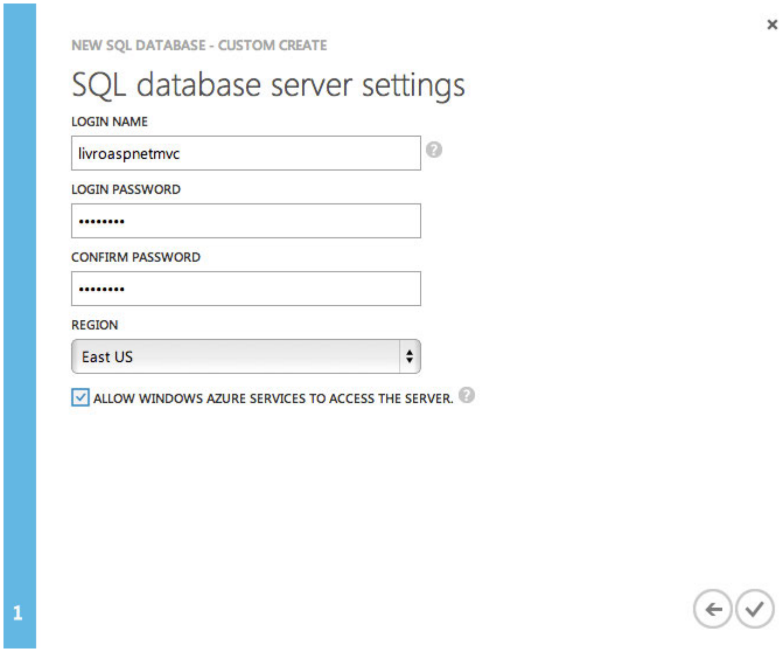


Figura 8.9: Segunda janela parametrizada na criação do banco de dados

Se tudo correu bem, você verá seu banco de dados criado na tela que exibe todos os recursos, conforme ilustra a figura 8.10.

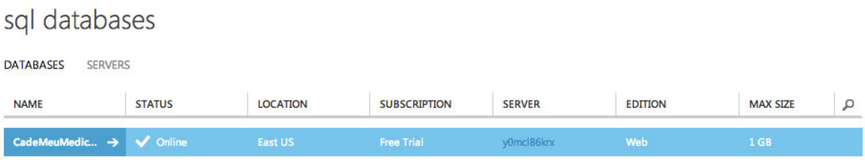


Figura 8.10: Banco de dados criado com sucesso

Antes de migrarmos o banco de dados do ambiente local para o online, precisamos liberar o acesso dos IP's permitidos ao banco de dados no Windows Azure. Claro, este bloqueio é feito por medida de segurança. Entretanto, para que possamos exemplificar o funcionamento, liberaremos todos os IP's. Para isso, no contexto do banco de dados recém-criado, clique na aba "Dashboard", localizada imediatamente abaixo do nome do banco de dados. Na janela que se abrirá, clique na opção "*Manage*

allowed IP addresses". Na tela que se abrirá, você deverá especificar o intervalo de IP's permitidos. Para liberar todos, informamos os valores "0.0.0.0" até "255.255.255.255", isto é, todos. A figura 8.11 ilustra esta parametrização.

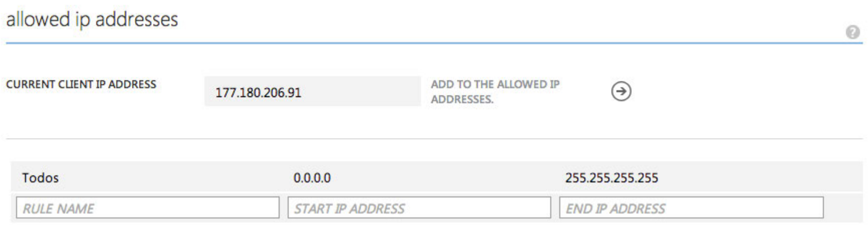


Figura 8.11: Especificando IPs permitidos no banco de dados do Azure

O passo seguinte consiste na migração do banco de dados local para o ambiente online. Existem diferentes formas para se realizar tal procedimento. Optamos por gerar um *script* utilizando o próprio *management studio* e, então, executá-lo neste novo ambiente. Você pode encontrar um bom texto sobre como gerar *scripts* a partir do banco de dados através deste link: http://bit.ly/gerandoscriptssql.

De posse dos dados de conexão com o banco de dados no Windows Azure, utilizarei o próprio *management studio* para realizar a migração, conforme ilustra a figura 8.12.

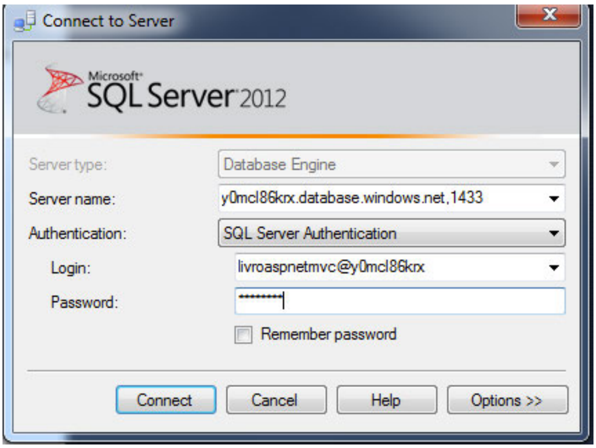


Figura 8.12: Conectando ao banco de dados no Windows Azure

Já conectado ao bando de dados, selecione o contexto do banco de dados "CadeMeuMedicoBD" e crie uma nova *query*. Copie o código gerado no parágrafo anterior, cole na aba recém-criada e execute-o. Se tudo correu bem, você estará visualizando à esquerda (na janela *Object Explorer*) algo semelhante ao que é apresentado pela figura 8.13.

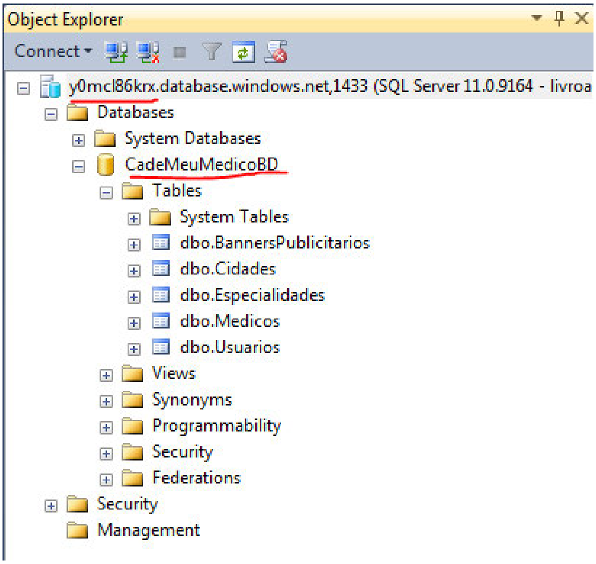


Figura 8.13: Migrando o banco de dados

Voltando à publicação...

Agora que já possuímos o banco de dados já publicado no Windows Azure, podemos seguir com o processo de publicação de nossa aplicação através do Visual Studio. Estávamos na etapa onde configurávamos o banco de dados, certo? O que precisaremos fazer agora é apontar para o banco que acabamos de migrar. A figura 8.14 apresenta a parametrização (que aponta para o banco online).

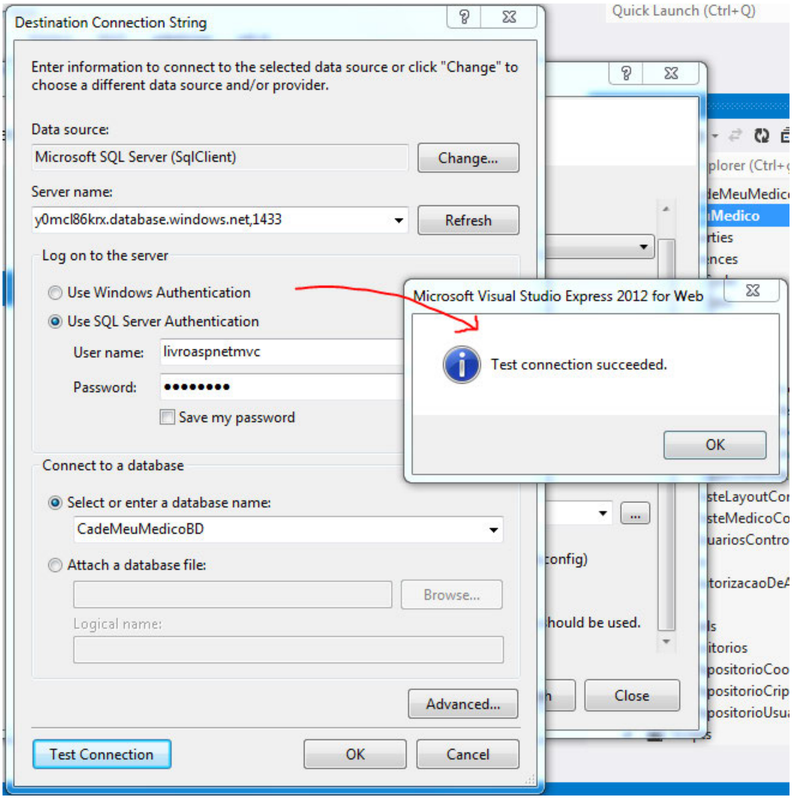


Figura 8.14: Apontando a aplicação para o banco de dados no Windows Azure

Na sequência, clique em "Ok" e, ao voltar para a janela anterior, clique em "*Next*". A última tela do *wizard* lhe apresentará a opção do *preview*. Esta etapa não é obrigatória mas, se você optar por sua execução, o Visual Studio irá fazer uma varredura no ambiente online e comparar com os arquivos do projeto local. Assim, você poderá ter a dimensão exata de quais arquivos serão publicados. Legal, não?! Ao realizar esta operação, o Visual Studio exibiu a tela apresentada pela figura 8.15 (semelhante à que você deverá estar visualizando).

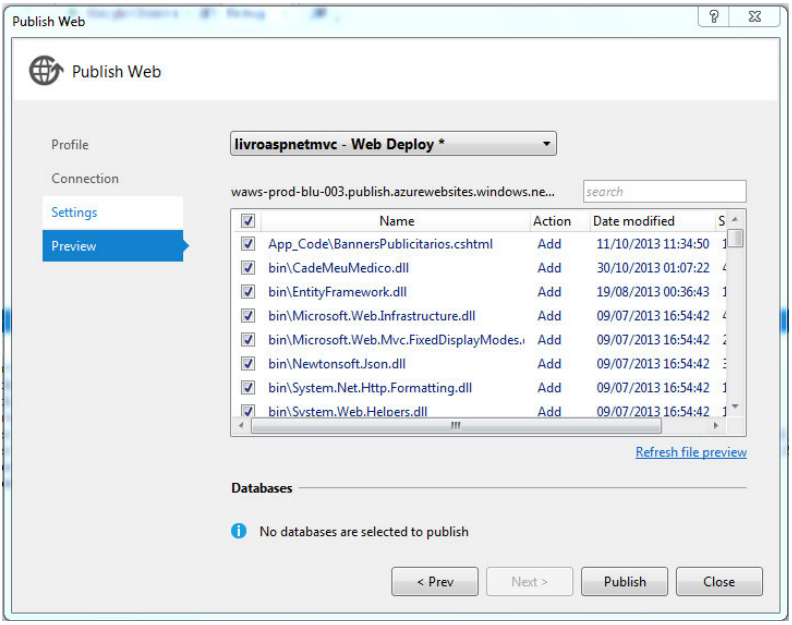


Figura 8.15: Preview dos arquivos que serão publicados

Por fim, clique no botão "*Publish*". Ao final, se tudo correr bem, o Visual Studio abrirá a aplicação já no ambiente online do Windows Azure, conforme ilustra a figura 8.16.

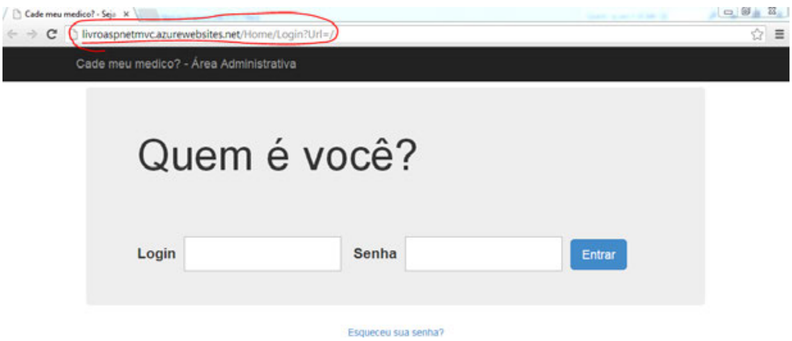


Figura 8.16: Aplicação publicada no Windows Azure Web Sites

Como você pode notar, o Windows Azure cria um subdomínio (*.azurewebsites.net) para a aplicação. Se quiséssemos personalizar o domínio (http://seudominio.

com.br) , bastaria nos dirigirmos até o site Registro.br e adicionarmos registros CNAME que apontariam para "http://livroaspnetmvc.azurewebsites.net/" .

8.7 Conclusão

Como você pôde perceber, existem diversas maneiras para que se possa publicar aplicações ASP.NET MVC. No próprio Windows Azure, existem pelo menos 3 maneiras distintas para executar este procedimento. Este capítulo apresentou apenas uma delas — a mais simples. Entretanto, é importante mencionar que a ideia geral não muda.

Esperamos que, ao completar a leitura deste livro, você possa ter ampliado seus horizontes em relação aos conhecimentos técnicos no quesito "desenvolvimento web" e, principalmente, considere utilizar ASP.NET MVC ao projetar aplicações futuras.

8.8 Código fonte e lista de discussão

O código fote da aplicação que construímos no livro pode ser encontrado no GitHub (http://bit.ly/mvc-livrocodigofonte) , utilize sempre que necessário para referência nos estudos e contribua com o código fonte, faça um fork e aguardamos seu pull request.

Além do código fonte um grupo de discussão (http://bit.ly/mvc-livrogrupodiscussao) foi criado para conversarmos sobre ASP.NET MVC e dúvidas referentes ao livro, aguardamos sua participação

Referências Bibliográficas

[1] Microsoft Virtual Academy. Introdução à business intelligence. 2012.

[2] Eric Enge. *A arte de SEO*. 2010.

[3] Steve Lydford. *Building ASP.NET Web Pages with Microsoft WebMatrix*. 2011.

[4] W3C Protocols. Hypertext transfer protocol - http/1.1. 1999.

[5] Fabrício Lopes Sanchez. Enriquecendo o visual de sua aplicação com master pages (blog). 2010.